UniSENAI

PROJETO APLICADO IV - SITE DE HOSPEDAGEM

ANDREY LUIZ PEREIRA

LUCA COMMUNELLO

LUIZ CARLOS SOUSA DA FONSECA

PAULO HENRIQUE BOCARDO

FLORIANÓPOLIS

2025

SUMÁRIO

1 INTRODUÇÃO

O presente documento técnico tem como objetivo apresentar informações abrangentes e detalhadas sobre o desenvolvimento, funcionalidades e estrutura da aplicação, que visa oferecer soluções para o gerenciamento simplificado de hospedagem na Pousada Quinta do Ypuã. Entre as necessidades da empresa, destacam-se o controle de hospedagens, gerenciamento de clientes, acomodações e funcionários (que são os usuários do sistema). Como restrições, destaca-se a dependência de integração com a internet e a necessidade de manter o custo da solução acessível.

Este documento abordará os requisitos funcionais e não funcionais do sistema, a modelagem da aplicação por meio de diagramas, a definição da API e seus respectivos endpoints, além dos testes unitários realizados e das tecnologias utilizadas ao longo do processo de desenvolvimento.

2 REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

O levantamento de requisitos é a construção de uma aplicação que atenda às necessidades dos usuários finais, esses requisitos podem ser funcionais e não funcionais, onde funcionais sendo referentes a funcionalidades específicas e não funcionais abordando aspectos voltados à performance, segurança, usabilidade, etc.

2.1 Requisitos funcionais

Os requisitos funcionais levantados com o objetivo de atender as necessidades da Pousada Quinta do Ypuã, abordam o gerenciamento de clientes, usuários, reservas, acomodações, autenticação e entre outros, conforme tabela abaixo:

Tabela 1 – Requisitos Funcionais do Sistema

ID	REQUISITO	DESCRIÇÃO	USUÁRIOS ENVOLVIDOS
RF01	Autenticação por Tela de Login	O sistema deve fornecer uma tela de login para autenticação dos usuários (administrador e funcionários), utilizando o CPF como login e senha.	Administrador, Funcionário
RF02	Cadastro de Usuário	O sistema deve permitir que o administrador crie usuários com permissão de funcionário, incluindo CPF, senha, nome, perfil, e-mail e status de habilitação.	Administrador
RF03	Visualização de Usuários	O sistema deve permitir que o administrador liste todos os usuários (administrador e funcionários), com dados como nome, perfil, e-mail e habilitação.	Administrador
RF04	Atualização de Permissão de Usuário	O sistema deve permitir que o administrador altere o status de habilitação dos usuários (funcionários), para garantir que ex-funcionários fiquem sem acesso.	Administrador
RF05	Atualização de Senha	O sistema deve permitir que os usuários (administrador e funcionários) atualizem sua senha, fornecendo uma nova senha e a confirmação da nova senha.	Administrador, Funcionário

RF06	Visualização de Acomodações	O sistema deve permitir que os usuários visualizem a lista de acomodações disponíveis.	Administrador, Funcionário
RF07	Criação de Acomodações	O sistema deve permitir que os usuários criem novas acomodações, com informações como nome, tipo, capacidade, preço, status habilitado, descrição e insira uma imagem.	Administrador, Funcionário
RF08	Atualização de Acomodações	O sistema deve permitir que os usuários atualizem as informações de acomodações existentes.	Administrador, Funcionário
RF09	Desabilitação de Acomodações	O sistema deve permitir que os usuários desabilitem acomodações, tornando-as indisponíveis para reservas.	Administrador, Funcionário
RF10	Visualização de Amenidades	O sistema deve permitir que os usuários visualizem a lista de amenidades disponíveis.	Administrador, Funcionário
RF11	Criação de Amenidades	O sistema deve permitir que os usuários criem novas amenidades, com o campo para nome.	Administrador, Funcionário
RF12	Atualização de Amenidades	O sistema deve permitir que os usuários atualizem as informações de amenidades existentes.	Administrador, Funcionário
RF13	Visualização de Reservas	O sistema deve permitir que os usuários visualizem todas as reservas de cada acomodação a partir do mês selecionado.	Administrador, Funcionário
RF14	Criação de Reservas	O sistema deve permitir que os usuários cadastrem novas reservas, com campos para data de entrada, data de saída, acomodação, cliente, status e valor total.	Administrador, Funcionário
RF15	Edição de Reservas	O sistema deve permitir que os usuários atualizem reservas existentes, como alterar o status de “Em andamento” para “Cancelado”.	Administrador, Funcionário
RF16	Controle de Disponibilidade em Tempo Real	O sistema deve exibir a disponibilidade de cada acomodação(habilitada) em tempo real, garantindo que a equipe visualize as ocupações.	Administrador, Funcionário
RF17	Dashboard Interativo com Power BI	O sistema deve apresentar um dashboard com um iframe do Power BI integrado, permitindo que o usuário visualize relatórios dinâmicos com dados extraídos diretamente do banco de dados. O dashboard inclui filtros interativos para que o usuário possa selecionar diferentes intervalos de datas, acomodação, ou qualquer	Administrador

		outro dado relevante para análise.	

Autor: Autoria própria (2025)

2.1 Requisitos não funcionais

Os requisitos não funcionais levantados para o sistema de hospedagens têm como objetivo garantir bom desempenho, usabilidade, segurança e entre outros, conforme tabela abaixo:

Tabela 2 – Requisitos Não Funcionais do Sistema

ID	REQUISITO	DESCRIÇÃO
RNF01	Autenticação por Tela de Login	O sistema deve ser intuitivo e de fácil navegação, com interface amigável e acessível, seguindo princípios de design centrado no usuário, incluindo feedback visual e usabilidade.
RNF02	Compatibilidade com Navegadores	A aplicação deve ser compatível com os principais navegadores, como Chrome, Firefox, Safari e Edge.
RNF03	Segurança	As senhas dos usuários e cpfs devem ser criptografadas antes de serem armazenadas, utilizando hashing seguro e salt. O sistema deve ter proteção contra ataques de força bruta e SQL Injection.
RNF04	Escalabilidade e Manutenção	O sistema deve ser fácil de manter, com código bem documentado e estrutura modular, permitindo atualizações e adições de novas funcionalidades.
RNF05	Baixo Custo de Infraestrutura	A solução deve utilizar tecnologias de baixo custo e preferencialmente de código aberto, minimizando os gastos com servidores e manutenção.
RNF06	Atualização Periódica de Dados no Dashboard	O sistema deve ser capaz de atualizar os dados exibidos no dashboard do Power BI a cada 5 minutos. O processo de atualização deve ser automatizado, sem intervenção manual, garantindo que os usuários sempre vejam dados recentes.

Autor: Autoria própria (2025)

3 MODELAGEM DO SISTEMA

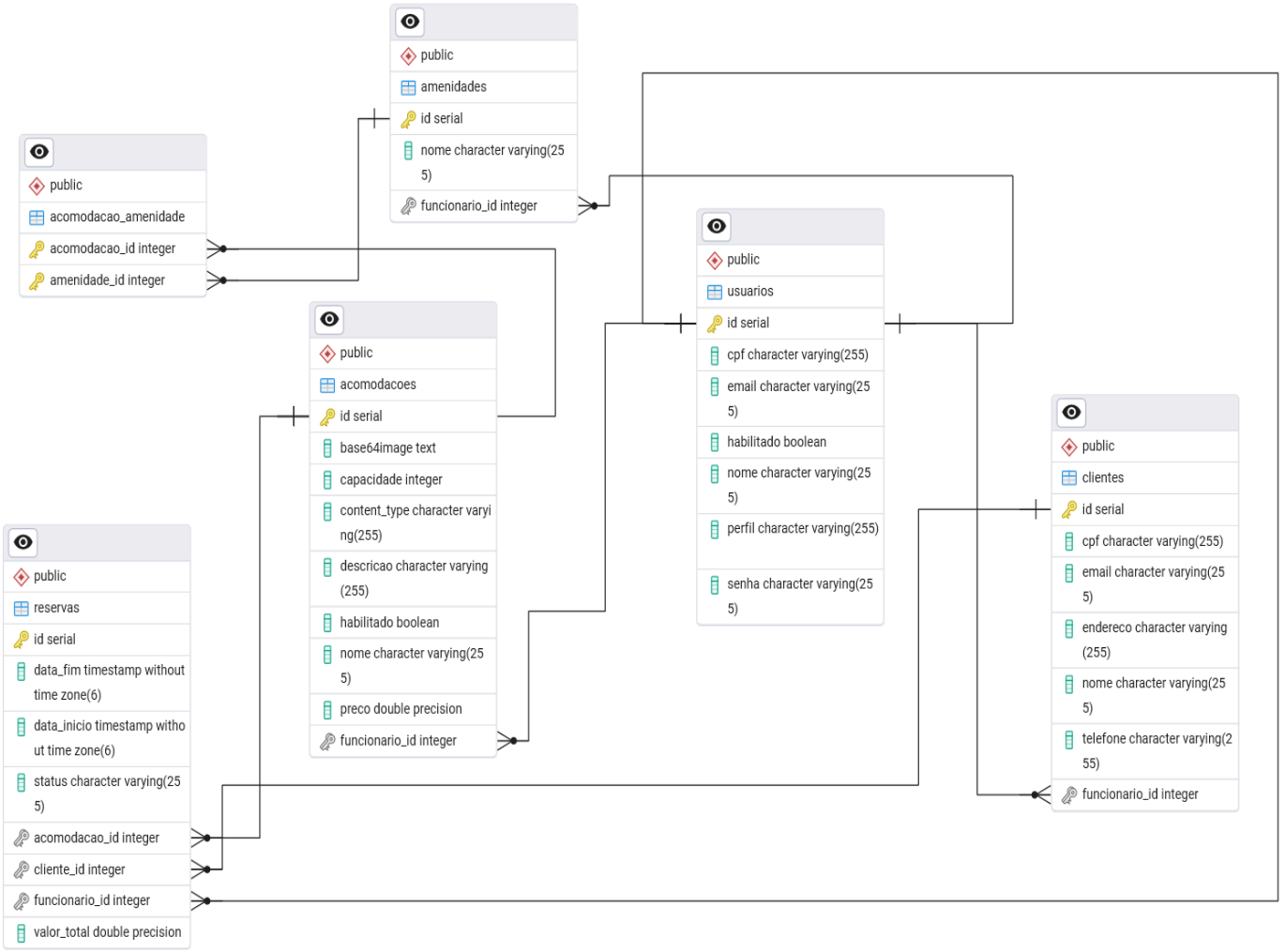
A modelagem do sistema é uma etapa essencial para representar de forma estruturada e visual o funcionamento e as interações do sistema de gerenciamento da Pousada Quinta do Ypuã. Esta seção apresenta os principais diagramas que ilustram a arquitetura e os processos do sistema, baseados nos requisitos funcionais e não funcionais definidos. Inclui o Diagrama de Classes, que descreve as entidades e suas relações; o Diagrama Entidade-Relacionamento (DER), que detalha a estrutura do banco de dados; o Diagrama de Casos de Uso, que define as funcionalidades disponíveis para os usuários (administrador e funcionário); o Diagrama de Atividades, que mapeia os fluxos de trabalho; o Diagrama de Sequência, que mostra a interação entre atores e componentes ao longo do tempo; o Diagrama de Componentes, que representa a organização dos módulos do sistema; e o Diagrama de Implantação, que ilustra a infraestrutura de deployment.

3.1 Diagrama de Classes

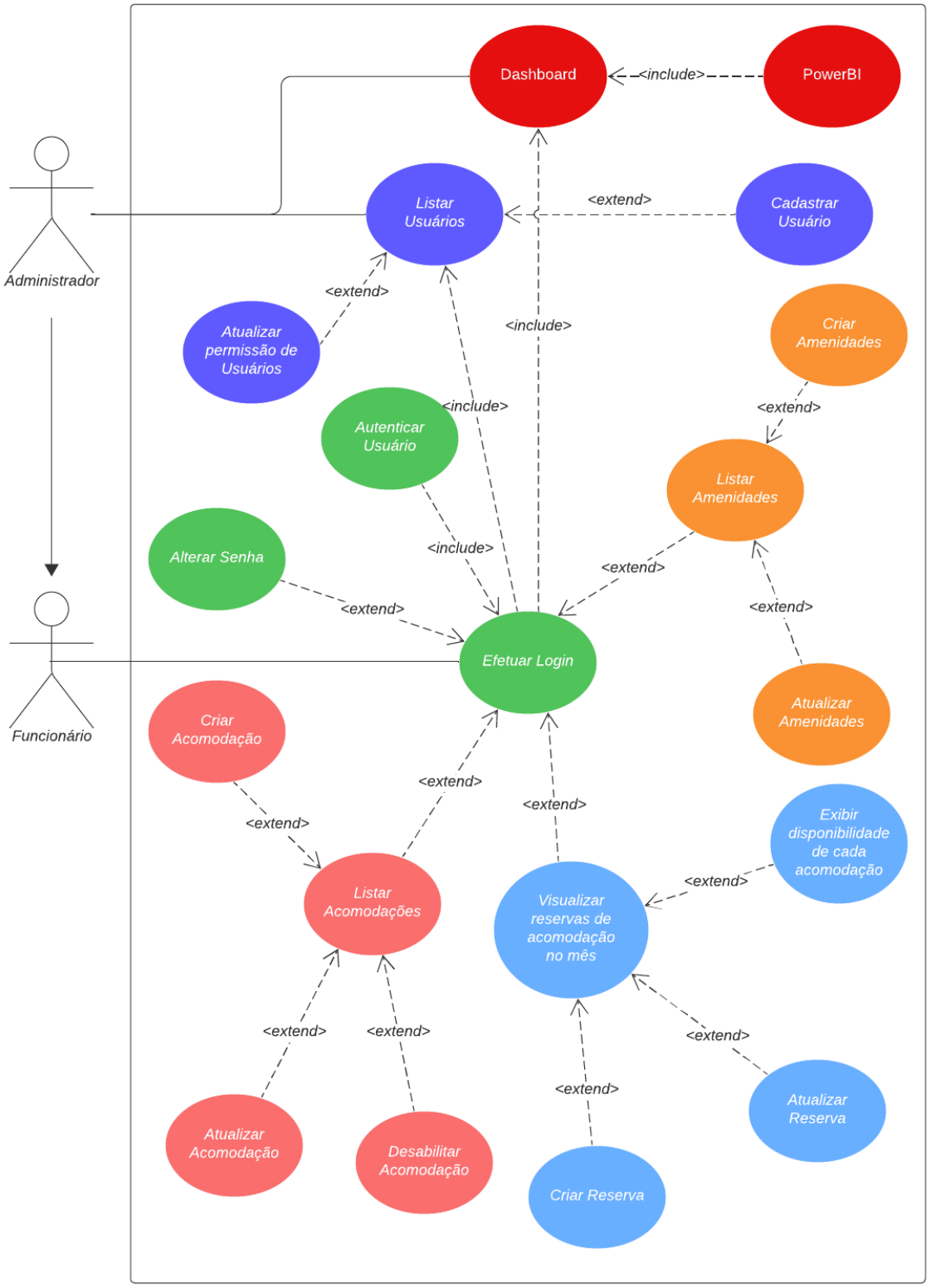
A figura completa do diagrama de classes pode ser acessada no link a seguir:

https://photos.google.com/share/AF1QipMUEs59LbosBmLbbBIORKcF5dxAdSov-OOZw-BxlCo8nlzrx8jhzfa-_hwltIlMoQ/photo/AF1QipMFc6qtCBL08WUFqtTtTrXJoitKHQ2gQ9HdVVuB?key=X25fdlc2VXZTU0ptRGNaVDg2ZDAxQURqVnFnZTZR

3.2 Diagrama Entidade-Relacionamento



3.3 Diagrama de Casos de Uso

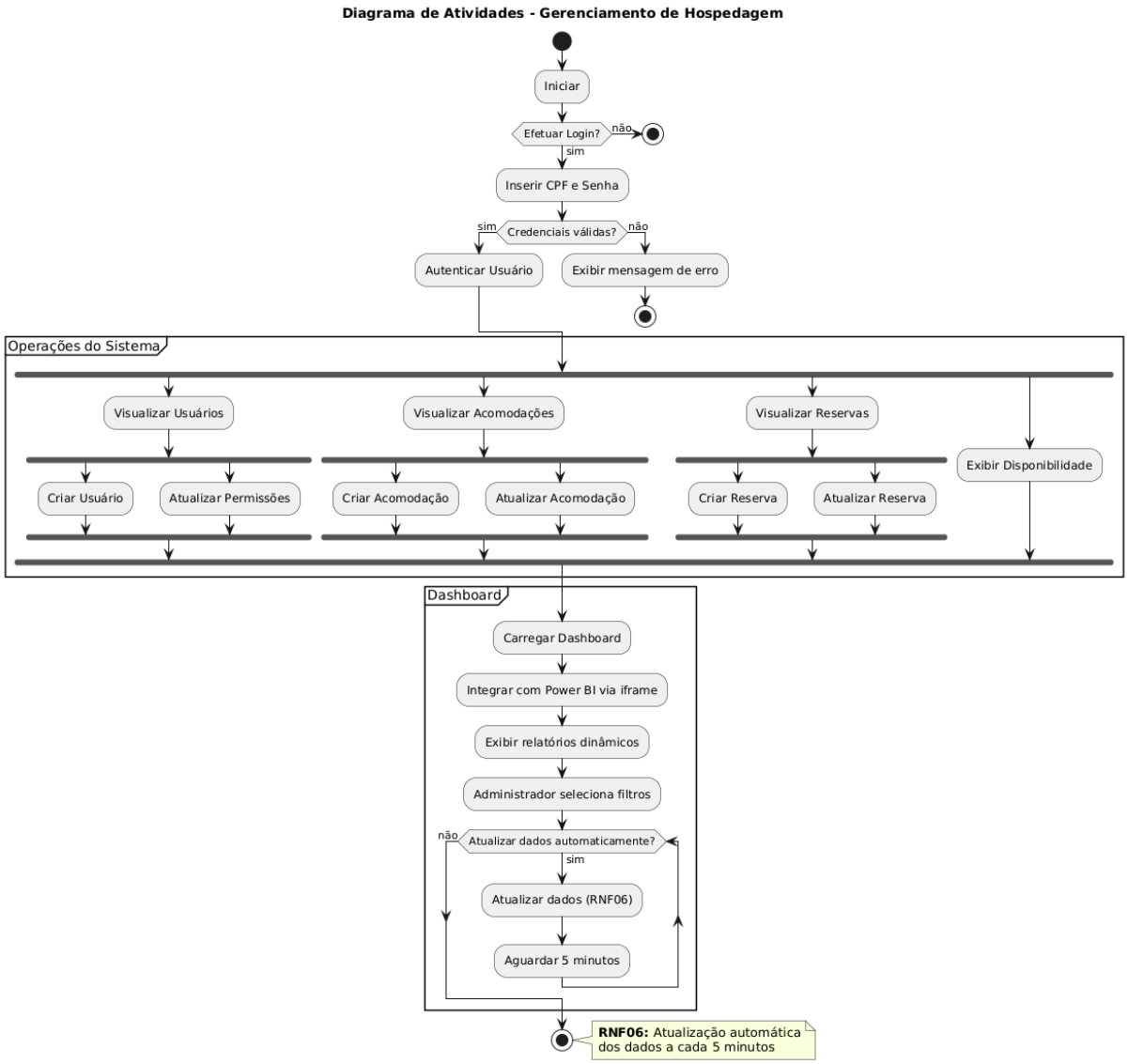


3.4 Diagrama de Casos de uso Expandido

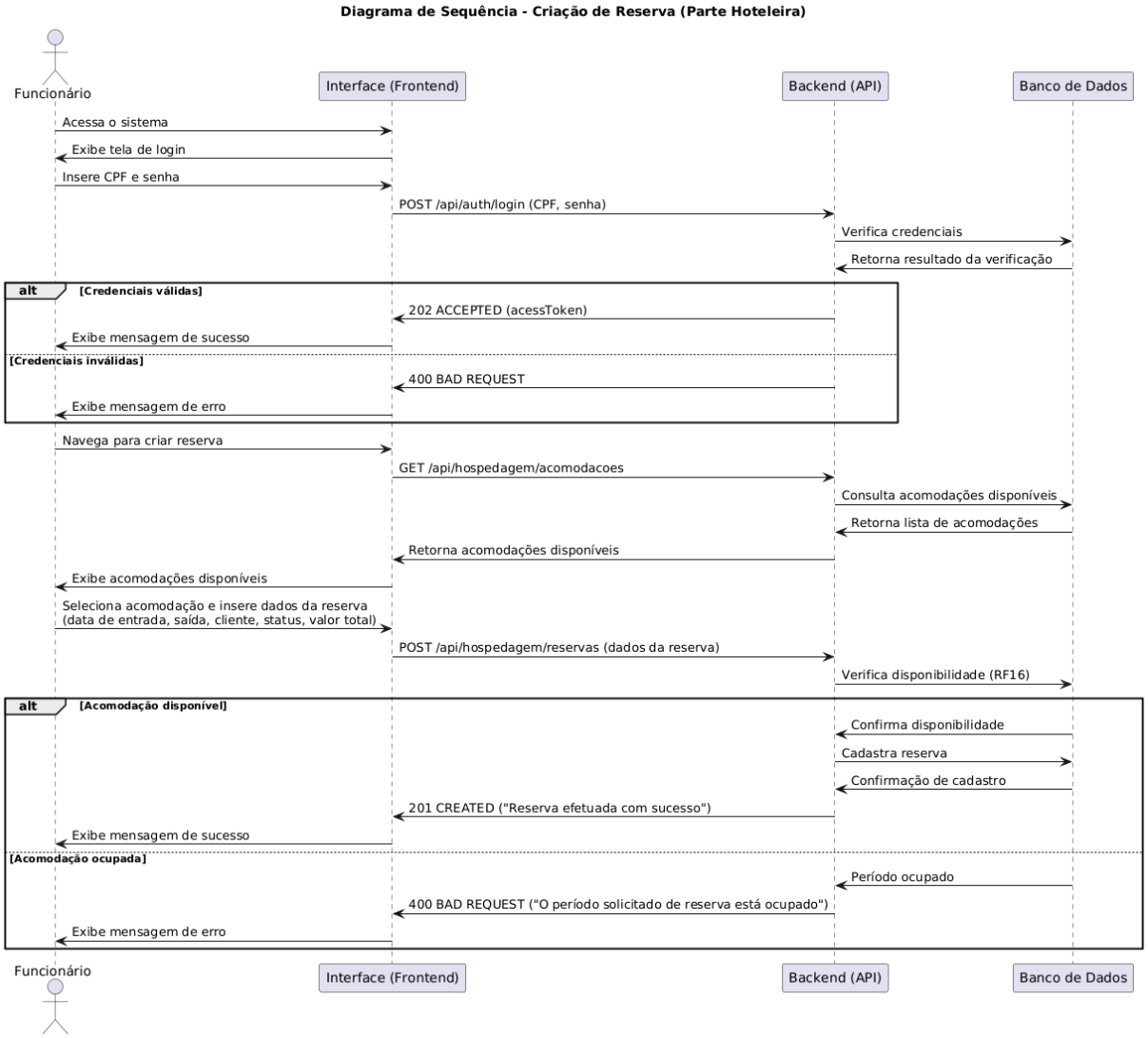
A figura completa do diagrama de casos de uso expandido pode ser acessada no link a seguir:

https://drive.google.com/file/d/1mEQNonQYIw6nA3_DCQf8Qt584rUpc3yj/view?usp=drive_link

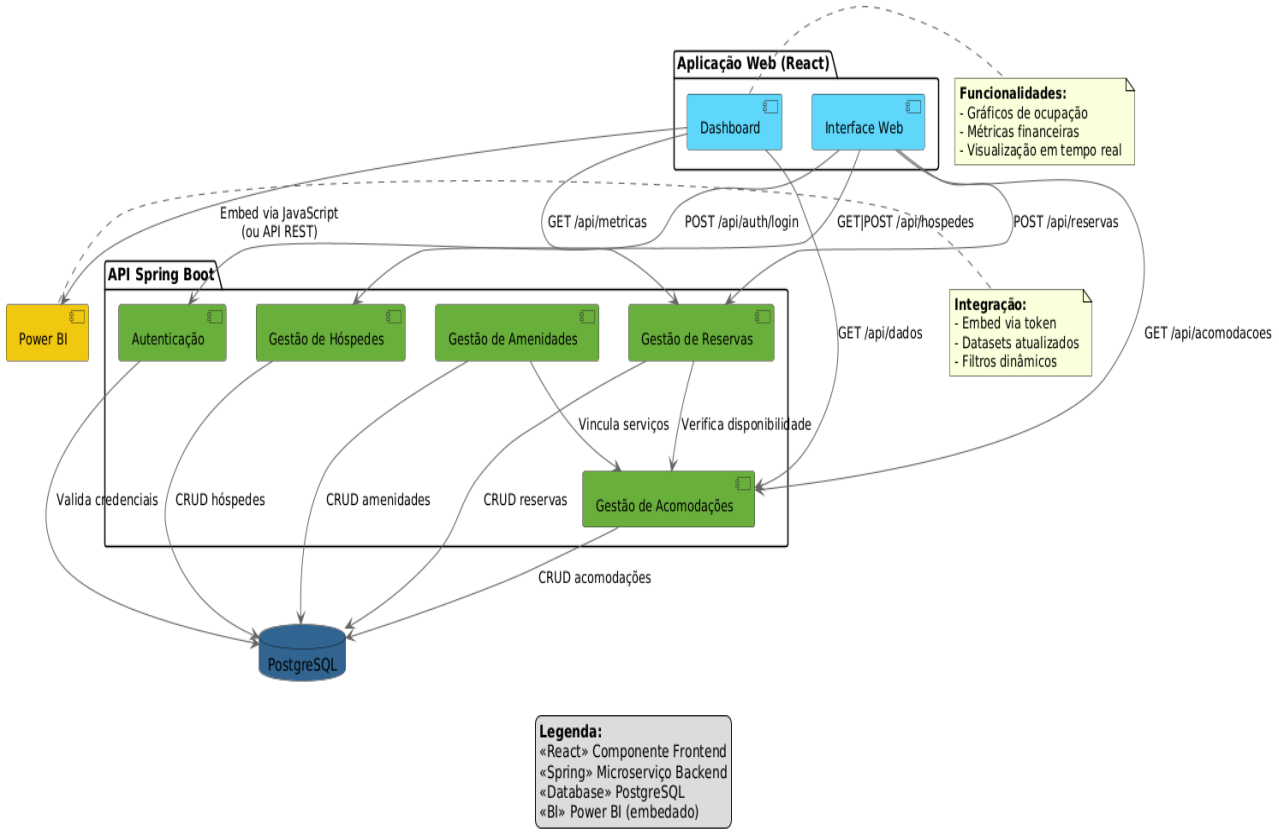
3.5 Diagrama de Atividades



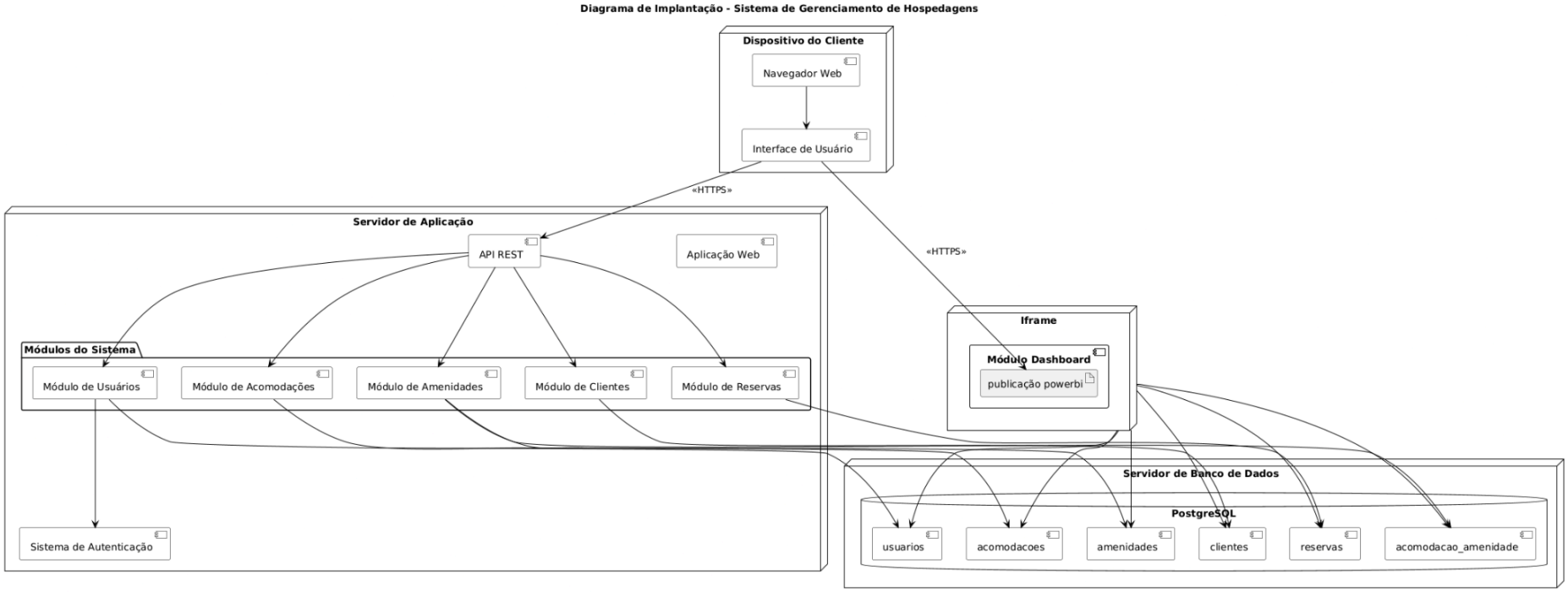
3.6 Diagrama de Sequência



3.7 Diagrama de Componentes



3.8 Diagrama de Implantação



4 API E ENDPOINTS

A aplicação possui 27 pontos de API construídos utilizando o Spring Boot, que abrange várias funcionalidades internas de gerenciamento, incluindo usuários, autenticação/autorização, clientes, acomodações, amenidades, reservas e outros. É importante destacar que todas as APIs são internas à aplicação e não há conexão com APIs externas.

A seguir, são apresentados os pontos de API e seus respectivos endpoints.

4.1 Operações para gerenciar usuários

4.1.1 Cadastrar Usuário

Endpoint

POST: /api/usuario/cadastrar

Authorization: Bearer {acessToken}

Descrição

Com um objeto usuário no corpo da requisição, efetua cadastro de usuário no banco de dados.

Parâmetros

- **Paths:**
    - Nenhum.

Body (JSON):

```
{
  "id": 0,
  "cpf": "string",
```

```
  "senha": "string",
  "perfil": "string",
  "nome": "string",
  "email": "string",
  "habilitado": true
}
```


Respostas

- **201 CREATED**


```
{
  "message": "Usuário cadastrado com sucesso."
}
```


- **400 BAD REQUEST**


```
{
  "message": "Não foi possível cadastrar usuário."
}
```


- **409 CONFLICT**


```
{
  "message": "O CPF fornecido é inválido."
}
```


4.1.2 Atualizar Dados do Usuário

Endpoint

PUT: /api/usuario/atualizar/{usuarioId}

Authorization: Bearer {acessToken}

Descrição

Com usuarioId como parâmetro e um objeto usuário no corpo da requisição, atualiza os dados do usuário que possui o valor do id do parâmetro.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.

Body (JSON):

```
{
  "id": 0,
  "cpf": "string",
  "senha": "string",
  "perfil": "string",
  "nome": "string",
  "email": "string",
  "habilitado": true
}
```

Respostas

- **200 OK**

```
{
  "message": "Usuário atualizado com sucesso."
```

```
}
```

- **404 NOT FOUND**

```
{
  "message": "Não foi possível atualizar o usuário."
}
```

4.1.3 Atualizar senha do Usuário

Endpoint

PUT: /api/usuario/atualizarSenha/{usuarioId}

Authorization: Bearer {acessToken}

Descrição

Com usuarioId como parâmetro e uma string com o valor de senha válido no corpo da requisição, atualiza a senha do usuário.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.

Body (JSON):

```
"string"
```

Respostas

- **200 OK**

```
{
  "message": "Senha atualizada com sucesso."
}
```

- **404 NOT FOUND**

```
{
  "message": "Não foi possível atualizar a senha."
}
```

- **409 CONFLICT**

```
{
  "message": "A senha fornecida é inválida."
}
```

4.1.4 Habilitar/Desabilitar Usuário

Endpoint

PUT: /api/usuario/lista/{usuarioId}/{habilitado}
Authorization: Bearer {acessToken}

Descrição

Com usuarioId e um boolean como parâmetros, altera a habilitação do usuário com id do parâmetro no banco de dados, habilitando e desabilitando acesso do usuário no sistema.

Parâmetros

- **Paths:**

- usuarioId (integer): ID do usuário.
- habilitado (boolean): true/false.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
{
  "message": "Credencial atualizada com sucesso."
}
```

- **404 NOT FOUND**

```
{
  "message": "Usuário com ID " + usuarioId + " não encontrado."
}
```

4.1.5 Listar Usuários

Endpoint

GET: /api/usuario/lista

Authorization: Bearer {acessToken}

Descrição

Recupera uma lista de usuários e seus respectivos dados do banco de dados.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
[
  {
    "id": 0,
    "cpf": "string",
    "senha": "string",
    "perfil": "string",
    "nome": "string",
    "email": "string",
    "habilitado": true
  }
]
```

- **400 BAD REQUEST**

```
{}
```

1.6 Recuperar dados de Usuário

Endpoint

GET: /api/usuario/lista/{usuarioId}

Authorization: Bearer {acessToken}

Descrição

Com usuarioId como parâmetro, recupera um objeto contendo dados do usuário.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
{
  "id": 0,
  "cpf": "string",
  "senha": "string",
  "perfil": "string",
  "nome": "string",
  "email": "string",
  "habilitado": true
}
```

- **400 BAD REQUEST**

```
{}
```

4.2 Autenticação

4.2.1 Autenticar e autorizar usuário

Endpoint

POST: /api/auth/login

Descrição

Com credenciais de CPF e senha efetuar autenticação na aplicação retornando um token para ser incluído no cabeçalho das demais requisições.

Parâmetros

- **Paths:**
    - Nenhum.

Body (JSON):

```
{
  "cpf": "string",
  "senha": "string"
}
```

Respostas

- **202 ACCEPTED**

```
{
  "acessToken": "string",
```

```
  "tokenType": "string"
}
```

- **400 BAD REQUEST**

```
{}
```

4.3 Operações para gerenciar as amenidades

4.3.1 Cadastrar Amenidade

Endpoint

POST: /api/hospedagem/{usuarioId}/amenidades
Authorization: Bearer {acessToken}

Descrição

Com usuarioId como parâmetro e um objeto amenidade no corpo da requisição, efetua cadastro da amenidade no banco de dados.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.

Body (JSON):

```
 {
   "nome": "string"
 }
```

Respostas

- **201 CREATED**

```
{
  "message": "Amenidade adicionada ao banco de dados."
}
```

- **400 BAD REQUEST**

```
{
  "message": "Já existe uma amenidade com este nome."
}
```

- **409 CONFLICT**

```
{
  "message": "Amenidade deve ter um nome válido."
}
```

4.3.2 Atualizar Dados da Amenidade

Endpoint

PUT: /api/hospedagem/{usuarioId}/amenidades/{amenidadeId}
Authorization: Bearer {acessToken}

Descrição

Com usuarioId e amenidadeId como parâmetros e um objeto amenidade no corpo da requisição, atualiza o usuário que possui o valor do id do parâmetro.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.

- amenidadeId (integer): ID da amenidade.

Body (JSON):

```
{
  “id”: 0,
  "nome": "string"
}
```

Respostas

- **200 OK**

```
{
  "message": "Amenidade atualizada com sucesso."
}
```

- **404 NOT FOUND**

```
{
  "message": "Amenidade não encontrada."
}
```

- **409 CONFLICT**

```
{
  "message": "Amenidade deve ter um nome válido."
}
```

4.3.3 Listar Amenidades

Endpoint

GET: /api/hospedagem/amenidades

Authorization: Bearer {acessToken}

Descrição

Recupera uma lista de amenidades e seus respectivos dados do banco de dados.

Parâmetros

- **Paths:**
    - Nenhum.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
[
  {
    "id": 0,
    "nome": "string"
  }
]
```

- **400 BAD REQUEST**

```
{}
```

4.3.4 Recuperar dados de Amenidade

Endpoint

GET: /api/hospedagem/amenidades/{amenidadeId}

Authorization: Bearer {acessToken}

Descrição

Com amenidadeId como parâmetro, recupera um objeto contendo dados da amenidade no banco de dados.

Parâmetros

- **Paths:**
    - amenidadeId (integer): ID da amenidade.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
{
  "id": 0,
  "nome": "string"
}
```

- **400 BAD REQUEST**

```
{}
```

4.4 Operações para gerenciar as acomodações

4.4.1 Cadastrar Acomodação

Endpoint

POST: /api/hospedagem/{usuarioId}/acomodacoes

Authorization: Bearer {acessToken}

Descrição

Com usuário Id como parâmetro e um objeto acomodação no corpo da requisição, efetua cadastro da acomodação no banco de dados.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.

Body (JSON):

```
{
  "id": 0,
  "nome": "string",
  "descricao": "string",
  "capacidade": 0,
  "funcionarioId": 0,
  "reservaId": 0,
  "preco": 0,
  "amenidades": [
    {
      "id": 0,
```

```
      "nome": "string"
    }
  ],
  "habilitado": true,
  "contentType": "string",
  "base64Image": "string"
}
```

Respostas

- **201 CREATED**

```
{
  "message": "Acomodação adicionada com sucesso."
}
```

- **400 BAD REQUEST**

```
{
  "message": "Os dados da acomodação estão incompletos."
}
```

4.4.2 Atualizar Dados da Acomodação

Endpoint

PUT: /api/hospedagem/{usuarioId}/acomodacoes/{acomodacaoId}

Authorization: Bearer {acessToken}

Descrição

Com usuário Id e acomodação Id como parâmetros e um objeto acomodação no corpo da requisição, atualiza o usuário que possui o valor do id do parâmetro.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.
    - acomodacaoId (integer): ID da acomodacao.

Body (JSON):

```
{
  "id": 0,
  "nome": "string",
  "descricao": "string",
  "capacidade": 0,
  "funcionarioId": 0,
  "reservaId": 0,
  "preco": 0,
  "amenidades": [
    {
      "id": 0,
      "nome": "string"
    }
  ],
  "habilitado": true,
  "contentType": "string",
  "base64Image": "string"
}
```

Respostas

- **200 OK**

```
{
  "message": "Acomodação atualizada com sucesso."
}
```

- **404 NOT FOUND**

```
{
  "message": "Amenidade com ID " + amenidade.getId() + " não encontrada."
}
```

- **404 NOT FOUND**

```
{
  "message": "Acomodação com ID " + acomodacaoId + " não encontrada."
}
```

4.4.3 Listar Acomodações

Endpoint

GET: /api/hospedagem/acomodacoes

Authorization: Bearer {acessToken}

Descrição

Recupera uma lista de acomodações e seus respectivos dados do banco de dados.

Parâmetros

- **Paths:**
    - Nenhum.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
[
  {
        "id": 0,
        "nome": "string",
        "descricao": "string",
        "capacidade": 0,
        "preco": 0,
        "habilitado": true,
        "contentType": "string",
        "base64Image": "string",
        "reservas": [
        {
        "id": 0,
        "cliente": "string",
        "dataInicio": "2025-05-05T22:29:51.963Z",
        "dataFim": "2025-05-05T22:29:51.963Z",
        "valorTotal": 0,
        "status": "Em andamento"
```

```
        }
        ],
        "amenidades": [
        {
        "id": 0,
        "nome": "string"
        }
        ]
  }
]
```


- **400 BAD REQUEST**


```
{}
```


4.4.4 Recuperar dados de Acomodação

Endpoint

GET: /api/hospedagem/acomodacoes/{acomodacaoId}

Authorization: Bearer {acessToken}

Descrição

Com acomodação Id como parâmetro, recupera um objeto contendo dados da acomodação no banco de dados.

Parâmetros

- **Paths:**
    - acomodacaoId (integer): ID da acomodacao.

Body (JSON):

```
{}
```

Respostas

- **200 OK**


```
{
  "id": 0,
  "nome": "string",
  "descricao": "string",
  "capacidade": 0,
  "preco": 0,
  "habilitado": true,
  "contentType": "string",
  "base64Image": "string",
  "reservas": [
        {
        "id": 0,
        "cliente": "string",
        "dataInicio": "2025-05-05T22:21:22.566Z",
        "dataFim": "2025-05-05T22:21:22.566Z",
        "valorTotal": 0,
        "status": "Em andamento"
        }
  ],
  "amenidades": [
        {
        "id": 0,
```

```
        "nome": "string"
        }
  ]
}
```

- **400 BAD REQUEST**

```
{}
```

4.4.5 Habilitar/Desabilitar da Acomodação

Endpoint

PUT: /api/hospedagem/acomodacoes/{acomodacaoId}/{habilitado}

Authorization: Bearer {acessToken}

Descrição

Com acomodação Id e habilitado(boolean) como parâmetros, atualiza no banco de dados o campo 'habilitado' da acomodação com o id definido no parâmetro.

Parâmetros

- **Paths:**
    - acomodacaoId (integer): ID da acomodação.
    - habilitado (boolean): true/false.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
{
  "message": "Estado da acomodação atualizado com sucesso."
}
```

- **404 NOT FOUND**

```
{
  "message": "Acomodação com ID " acomodacaoId" não encontrado."
}
```

4.5 Operações para gerenciar as reservas

4.5.1 Cadastrar Reservas

Endpoint

POST: /api/hospedagem/reservas

Authorization: Bearer {acessToken}

Descrição

Com um objeto reserva no corpo da requisição, efetua cadastro da reserva no banco de dados.

Parâmetros

- **Paths:**
    - Nenhum.

Body (JSON):

```
{
  "id": 0,
  "funcionarioId": 0,
  "clienteId": 0,
  "acomodacaoId": 0,
  "dataInicio": "2025-03-21T17:48:20.719Z",
  "dataFim": "2025-03-21T17:48:20.719Z",
  "status": "Em andamento",
  "valorTotal": 0
}
```


Respostas

- **201 CREATED**


```
{
  "message": "Reserva efetuada com sucesso."
}
```


- **400 BAD REQUEST**


```
{
  "message": "O período solicitado de reserva está ocupado."
}
```


- **404 NOT FOUND**


```
{
```

```
  "message": "Dados obrigatórios não fornecidos."
}
```

- **404 NOT FOUND**

```
{
  "message": "Usuário criador não encontrado."
}
```

- **404 NOT FOUND**

```
{
  "message": "Cliente não encontrado."
}
```

- **404 NOT FOUND**

```
{
  "message": "Acomodação não encontrada."
}
```

- **409 CONFLICT**

```
{
  "message": “O período de reserva não foi definido corretamente."
}
```

4.5.2 Atualizar Dados da Reserva

Endpoint

PUT: /api/hospedagem/reservas/{reservaId}

Authorization: Bearer {acessToken}

Descrição

Com reserva Id como parâmetro e um objeto reserva no corpo da requisição, atualiza a reserva que possui o valor do id do parametro.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.
    - acomodacaoId (integer): ID da acomodacao.

Body (JSON):

```
{
  "id": 0,
  "funcionarioId": 0,
  "clienteId": 0,
  "acomodacaoId": 0,
  "dataInicio": "2025-05-05T22:21:22.578Z",
  "dataFim": "2025-05-05T22:21:22.578Z",
  "status": "Em andamento",
  "valorTotal": 0
}
```

Respostas

- **200 OK**

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

- **400 BAD REQUEST**

```
{
  "message": "O período solicitado de reserva está ocupado."
}
```

- **404 NOT FOUND**

```
{
  "message": "Dados obrigatórios não fornecidos."
}
```

- **404 NOT FOUND**

```
{
  "message": "Reserva não encontrada."
}
```

- **404 NOT FOUND**

```
{
  "message": "Usuário criador não encontrado."
}
```

- **404 NOT FOUND**

```
{
  "message": "Cliente não encontrado."
}
```

- **404 NOT FOUND**

```
{
  "message": "Acomodação não encontrada."
}
```

- **409 CONFLICT**

```
{
  "message": “O período de reserva não foi definido corretamente."
}
```

4.5.3 Listar Reservas

Endpoint

GET: /api/hospedagem/reservas

Authorization: Bearer {acessToken}

Descrição

Recupera uma lista de reservas e seus respectivos dados do banco de dados.

Parâmetros

- **Paths:**
  - Nenhum.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
[
  {
    "id": 0,
    "funcionarioId": 0,
    "clienteId": 0,
    "acomodacaoId": 0,
    "dataInicio": "2025-03-21T18:04:41.426Z",
    "dataFim": "2025-03-21T18:04:41.426Z",
    "status": "Em andamento",
    "valorTotal": 0
  } ]
```

- **400 BAD REQUEST**

```
{}
```

4.5.4 Recuperar dados das Reservas

Endpoint

GET: /api/hospedagem/reservas/{reservaId}

Authorization: Bearer {acessToken}

Descrição

Com reserva Id como parâmetro, recupera um objeto contendo dados da reserva no banco de dados.

Parâmetros

- **Paths:**
    - reservaId (integer): ID da reserva.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
{
  "id": 0,
  "funcionarioId": 0,
  "clienteId": 0,
  "acomodacaoId": 0,
  "dataInicio": "2025-03-21T18:18:55.042Z",
  "dataFim": "2025-03-21T18:18:55.042Z",
  "status": "Em andamento",
  "valorTotal": 0
}
```

- **400 BAD REQUEST**

```
{}
```

4.6 Operações para gerenciar a agenda e calendário

4.6.1 Gerar uma agenda do dia atual com a disponibilidade de cada acomodação

Endpoint

GET: /api/hospedagem/agenda/{dia}

Authorization: Bearer {acessToken}

Descrição

Com a data do dia como parâmetro, gera uma lista com todas as acomodações, identificando a disponibilidade ou não da acomodação.

Parâmetros

- **Paths:**
    - dia (LocalDateTime): dia.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
[
  {
    "reservaId": 0,
    "clienteNome": "string",
    "clienteEmail": "string",
```

```
    "clienteTelefone": "string",
    "funcionarioNome": "string",
    "acomodacaoNome": "string",
    "reservaStatus": "string",
    "acomodacaoId": 0,
    "dataInicio": "2025-03-21T18:28:23.674Z",
    "dataFim": "2025-03-21T18:28:23.674Z",
    "valorTotal": 0
  }
]
```


- **400 BAD REQUEST**


```
{}
```


4.6.2 Recupera uma lista de agendamentos efetuados no mês de acordo com a acomodação

Endpoint

GET: /api/hospedagem/agenda/{acomodacaoId}/{mes}
Authorization: Bearer {acessToken}

Descrição

Com os parâmetros mês e o id da acomodação, vai gerar uma agenda do mês, informando as reservas realizadas naquela acomodação no mês.

Parâmetros

- **Paths:**
    - mes (LocalDateTime): mês.

- acomodacaoId(integer) ID da acomodação.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
[
  {
    "reservaId": 0,
    "clienteNome": "string",
    "clienteEmail": "string",
    "clienteTelefone": "string",
    "funcionarioNome": "string",
    "acomodacaoNome": "string",
    "reservaStatus": "string",
    "acomodacaoId": 0,
    "dataInicio": "2025-03-21T18:28:23.674Z",
    "dataFim": "2025-03-21T18:28:23.674Z",
    "valorTotal": 0
  }
]
```

- **400 BAD REQUEST**

```
{}
```

4.6.3 Gera calendário do mês com a disponibilidade da acomodação em cada dia

Endpoint

GET: /api/hospedagem/agenda/datas/{acomodacaoId}/{mes}

Authorization: Bearer {acessToken}

Descrição

Com os parâmetros mês e o id da acomodação, gerá um calendário do mês, informando cada dia do mês e um boolean para identificar se o dia a acomodação está ocupada ou não.

Parâmetros

- **Paths:**
    - dia (LocalDateTime): dia.
    - acomodacaoId (integer): ID da acomodação

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
[
  {
    "data": "2025-03-21T18:34:58.709Z",
    "ocupado": true,
    "reservaId": 0
```

```
  }
]
```

- **400 BAD REQUEST**

```
{}
```

4.7 Operações para gerenciar clientes

4.7.1 Cadastrar Clientes

Endpoint

POST: /api/hospedagem/{usuarioId}/clientes

Authorization: Bearer {acessToken}

Descrição

Com usuarioId como parâmetro e um objeto cliente no corpo da requisição, efetua cadastro do cliente no banco de dados.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.

Body (JSON):

```
{
  "id": 0,
  "cpf": "string",
  "nome": "string",
  "email": "string",
```

```
  "telefone": "string",
  "endereco": "string",
  "funcionarioId": 0,
  "reservaId": 0
}
```

Respostas

- **201 CREATED**

```
{
  "message": "Cliente cadastrado com sucesso."
}
```

- **409 CONFLICT**

```
{
  "message": "Formulário está incompleto, preencha todos os dados."
}
```

- **409 CONFLICT**

```
{
  "message": "Cliente com o CPF inválido."
}
```

- **409 CONFLICT**

```
{
  "message": "Funcionário não existe."
```

```
}
```

4.7.2 Atualizar Clientes

Endpoint

PUT: /api/hospedagem/{usuarioId}/clientes/{clienteId}

Authorization: Bearer {acessToken}

Descrição

Com usuarioId como parâmetro e um objeto cliente no corpo da requisição, efetua atualização do cliente no banco de dados.

Parâmetros

- **Paths:**
    - usuarioId (integer): ID do usuário.
    - clienteId (integer): ID do cliente.

Body (JSON):

```
{
  "id": 0,
  "cpf": "string",
  "nome": "string",
  "email": "string",
  "telefone": "string",
  "endereco": "string",
  "funcionarioId": 0,
  "reservaId": 0
```

```
}
```

Respostas

- **200 OK**

```
{
  "message": "Cliente atualizado com sucesso."
}
```

- **409 CONFLICT**

```
{
  "message": "Formulário está incompleto, preencha todos os dados."
}
```

- **409 CONFLICT**

```
{
  "message": "Cliente não cadastrado no sistema."
}
```

- **409 CONFLICT**

```
{
  "message": "Funcionário não cadastrado no sistema."
}
```

4.7.3 Listar Clientes

Endpoint

GET: /api/hospedagem/clientes

Authorization: Bearer {acessToken}

Descrição

Recupera uma lista de clientes e seus respectivos dados do banco de dados.

Parâmetros

- **Paths:**
  - Nenhum.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
[
  {
        "id": 0,
        "cpf": "string",
        "nome": "string",
        "email": "string",
        "telefone": "string",
        "endereco": "string",
```

```
        "reservas": null
  }
]
```

- **400 BAD REQUEST**

```
{}
```

4.7.4 Recuperar dados das Reservas

Endpoint

GET: /api/hospedagem/clientes/{clienteId}

Authorization: Bearer {acessToken}

Descrição

Com clienteId como parâmetro, recupera um objeto contendo dados do cliente no banco de dados.

Parâmetros

- **Paths:**
    - clienteId (integer): ID do cliente.

Body (JSON):

```
{}
```

Respostas

- **200 OK**

```
{
    "id": 0,
    "cpf": "string",
    "nome": "string",
     "email": "string",
     "telefone": "string",
     "endereco": "string",
     "reservas": null
}
```


- **400 BAD REQUEST**


```
{}
```

5 TESTES UNITÁRIOS

Testes unitários são componentes essenciais no processo de desenvolvimento de software, pois garantem que partes individuais do sistema — chamadas de *unidades* — funcionem conforme o esperado de forma isolada. No contexto desta aplicação, cada teste foi elaborado para verificar o comportamento de métodos específicos nos serviços principais, simulando interações com dependências através de *mocks* com a biblioteca Mockito.

Esses testes foram desenvolvidos com **JUnit 5** e estão localizados no seguinte caminho do projeto:

```
src/test/java/com/senai/api/services/
```

5.1 JUSTIFICATIVA DAS ESCOLHAS DOS TESTES UNITÁRIOS

A escolha do JUnit e do Mockito se baseou na capacidade dessas ferramentas de promover agilidade e clareza durante o desenvolvimento dos testes.

O JUnit oferece uma estrutura padronizada que facilita a organização dos casos de teste e a leitura dos resultados, tornando o processo mais intuitivo tanto para desenvolvedores quanto para revisores de código. Já o Mockito foi selecionado por permitir simulações precisas de dependências externas, o que viabilizou a criação de cenários realistas sem a necessidade de acionar serviços reais ou configurações complexas.

Essa abordagem contribuiu para um desenvolvimento mais fluido e para a redução de gargalos durante as fases de validação das funcionalidades.

5.2 DESCRIÇÃO E EXECUÇÃO DOS TESTES

A estrutura dos testes está organizada em arquivos distintos, seguindo a separação por serviço. Cada classe de teste tem como responsabilidade validar as principais operações disponíveis no respectivo serviço, como criação, edição, recuperação e manipulação de dados. Atualmente, os testes estão distribuídos em seis arquivos, cobrindo os seguintes serviços:

- **Acomodação**
- **Agenda**
- **Amenidade**
- **Cliente**
- **Reserva**
- **Usuário**

Cada seção a seguir representa um dos arquivos de teste.

5.2.1 AcomodacaoServiceTest.java

Este serviço é responsável pelas operações relacionadas à gestão de acomodações, como cadastro, edição, alteração de status (ativo/inativo) e consultas.

Testes implementados

testEditar

- **Descrição:** Avalia o método responsável por modificar os dados de uma acomodação existente.
- **Objetivo:** Garantir que a operação de edição seja processada corretamente, sem lançar exceções.

```
    @Test
    void testEditar() {
        AcomodacaoDto acomodacaoDto = new AcomodacaoDto();
                        when(acomodacaoService.editar(acomodacaoDto,   1,
1)).thenReturn(ResponseEntity.ok().build());
                                              assertDoesNotThrow(()      ->
acomodacaoService.editar(acomodacaoDto, 1, 1));
    }
```

testCadastrar

- **Descrição:** Valida o método de registro de uma nova acomodação, associando-a a um usuário.
- **Objetivo:** Assegurar que o serviço aceite os dados válidos e retorne uma resposta de sucesso (HTTP 200).

```
    @Test
    void testCadastrar() {
        AcomodacaoDto acomodacaoDto = new AcomodacaoDto();
                        when(acomodacaoService.cadastrar(acomodacaoDto,
1)).thenReturn(ResponseEntity.ok().build());
                                            assertDoesNotThrow(()      ->
acomodacaoService.cadastrar(acomodacaoDto, 1));
    }
```

testHabilitadoDesabilitado

- **Descrição:** Testa a funcionalidade de alteração do estado de disponibilidade de uma acomodação.
- **Objetivo:** Confirmar que o serviço trata adequadamente a requisição de habilitação/desabilitação.

```
    @Test
    void testHabilitadoDesabilitado() {
                       when(acomodacaoService.habilitadoDesabilitado(1,
true)).thenReturn(ResponseEntity.ok().build());
                                            assertDoesNotThrow(()      ->
acomodacaoService.habilitadoDesabilitado(1, true));
    }
```

testRecuperarAcomodacao

- **Descrição:** Verifica a recuperação de uma única acomodação, com base em seu identificador.
- **Objetivo:** Confirmar que o método retorna uma instância de Acomodação e responde com status 200.

```
    @Test
    void testRecuperarAcomodacao() {

when(acomodacaoService.recuperarAcomodacao(1)).thenReturn(ResponseEntit
y.ok(new Acomodacao()));
                                          assertDoesNotThrow(()      ->
acomodacaoService.recuperarAcomodacao(1));
    }
```

testRecuperarAcomodacoes

- **Descrição:** Testa a listagem de todas as acomodações registradas no sistema.
- **Objetivo:** Garantir que o serviço retorne uma coleção, mesmo que vazia, sem falhas.

```
    @Test
    void testRecuperarAcomodacoes() {

when(acomodacaoService.recuperarAcomodacoes()).thenReturn(ResponseEntit
y.ok(Collections.emptyList()));
```

```
                                                assertDoesNotThrow(()      ->
acomodacaoService.recuperarAcomodacoes());

    }

}
```

5.2.2 AgendaServiceTest.java

Este serviço trata da geração de agendas e calendários, tanto em tempo real quanto mensais, com base em datas e identificação do usuário.

Testes implementados

testGerarCalendarioMensal

- **Descrição:** Verifica se a geração de um calendário mensal com base na data atual e ID do usuário ocorre corretamente.
- **Objetivo:** Validar o retorno de uma lista de dados, assegurando que o método execute sem erros.

```
    @Test

    void testGerarCalendarioMensal() {

        LocalDateTime now = LocalDateTime.now();

                            when(agendaService.gerarCalendarioMensal(now,
1)).thenReturn(Collections.emptyList());

                                                assertDoesNotThrow(()      ->
agendaService.gerarCalendarioMensal(now, 1));

    }
```

testGerarAgendaMensal

- **Descrição:** Testa a criação de uma agenda mensal para um usuário específico.
- **Objetivo:** Confirmar que o método retorna uma lista adequada e não lança exceções.

```
    @Test
    void testGerarAgendaMensal() {
        LocalDateTime now = LocalDateTime.now();
                                when(agendaService.gerarAgendaMensal(now,
1)).thenReturn(Collections.emptyList());
          assertDoesNotThrow(() -> agendaService.gerarAgendaMensal(now,
1));
    }
```

testGerarAgendaTempoReal

- **Descrição:** Avalia a geração de uma agenda em tempo real com base na data corrente.
- **Objetivo:** Verificar se o método responde com sucesso e entrega uma lista.

```
    @Test
    void testGerarAgendaTempoReal() {
        LocalDateTime now = LocalDateTime.now();

when(agendaService.gerarAgendaTempoReal(now)).thenReturn(Collections.em
ptyList());
                                          assertDoesNotThrow(()      ->
agendaService.gerarAgendaTempoReal(now));
    }
```

```
}
```

5.2.3 ClienteServiceTest.java

Responsável por operações relacionadas ao gerenciamento de clientes, como cadastro, edição e recuperação de dados.

Testes implementados

testCadastrar

- **Descrição:** Avalia o método que registra um novo cliente vinculado a um usuário.
- **Objetivo:** Garantir que a requisição de cadastro seja aceita com dados válidos.

```
  @Test
    void testCadastrar() throws Exception {
        ClienteDto clienteDto = new ClienteDto();
                                when(clienteService.cadastrar(clienteDto,
1)).thenReturn(ResponseEntity.ok().build());
          assertDoesNotThrow(() -> clienteService.cadastrar(clienteDto,
1));
    }
```

testEditar

- **Descrição:** Testa o processo de atualização das informações de um cliente já existente.

- **Objetivo:** Validar que o método executa corretamente e retorna a resposta adequada.

```
    @Test
    void testEditar() throws Exception {
        ClienteDto clienteDto = new ClienteDto();
                            when(clienteService.editar(clienteDto,    1,
1)).thenReturn(ResponseEntity.ok().build());
          assertDoesNotThrow(() -> clienteService.editar(clienteDto, 1,
1));
    }
```

testRecuperarClientes

- **Descrição:** Testa a listagem de todos os clientes registrados no sistema.
- **Objetivo:** Assegurar que uma lista (mesmo vazia) seja retornada sem exceções.

```
    @Test
    void testRecuperarClientes() {

when(clienteService.recuperarClientes()).thenReturn(ResponseEntity.ok(C
ollections.emptyList()));
        assertDoesNotThrow(() -> clienteService.recuperarClientes());
    }
```

testRecuperarCliente

- **Descrição:** Avalia a recuperação de um cliente específico com base em seu ID.
- **Objetivo:** Confirmar que o método retorna uma instância de Cliente e responde com sucesso.

```
    @Test
    void testRecuperarCliente() {

when(clienteService.recuperarCliente(1)).thenReturn(ResponseEntity.ok(n
ew Cliente()));
        assertDoesNotThrow(() -> clienteService.recuperarCliente(1));
    }
}
```

5.2.4 AmenidadeServiceTest.java

O serviço de amenidades gerencia itens ou características adicionais oferecidas em acomodações (ex.: café da manhã, estacionamento, Wi-Fi etc.). Os testes verificam operações básicas como cadastro, edição e recuperação de dados.

Testes implementados
testCadastrar

- **Descrição:** Válida a criação de uma nova amenidade associada a um usuário.
- **Objetivo:** Garantir que o serviço aceita os dados fornecidos e retorna uma resposta de sucesso.

```
    @Test
    void testCadastrar() {
```

```
        AmenidadeDto amenidadeDto = new AmenidadeDto();

                            when(amenidadeService.cadastrar(amenidadeDto,
1)).thenReturn(ResponseEntity.ok().build());

                                                assertDoesNotThrow(()      ->
amenidadeService.cadastrar(amenidadeDto, 1));

    }
```

testEditar

- **Descrição:** Avalia o processo de modificação de uma amenidade existente.
- **Objetivo:** Certificar que o método pode ser executado sem erros e com retorno HTTP 200.

```
    @Test

    void testEditar() {

        AmenidadeDto amenidadeDto = new AmenidadeDto();

                          when(amenidadeService.editar(amenidadeDto,   1,
1)).thenReturn(ResponseEntity.ok().build());

         assertDoesNotThrow(() -> amenidadeService.editar(amenidadeDto,
1, 1));

    }
```

testRecuperarAmenidades

- **Descrição:** Testa a listagem de todas as amenidades disponíveis no sistema.
- **Objetivo:** Assegurar que o método retorna uma coleção, mesmo que vazia, sem lançar exceções.

```
    @Test
```

```
    void testRecuperarAmenidades() {

when(amenidadeService.recuperarAmenidades()).thenReturn(ResponseEntity.
ok(Collections.emptyList()));

                                                assertDoesNotThrow(()      ->
amenidadeService.recuperarAmenidades());

    }
```

testRecuperarAmenidade

- **Descrição:** Verifica a recuperação de uma amenidade específica por ID.
- **Objetivo:** Confirmar que o serviço retorna corretamente um objeto do tipo Amenidade.

```
    @Test

    void testRecuperarAmenidade() {

when(amenidadeService.recuperarAmenidade(1)).thenReturn(ResponseEntity.
ok(new Amenidade()));

                                                assertDoesNotThrow(()      ->
amenidadeService.recuperarAmenidade(1));

    }

}
```

5.2.5 ReservaServiceTest.java

O serviço de reservas trata da criação, edição, verificação de disponibilidade e consulta de reservas para acomodações. Os testes abrangem tanto funcionalidades CRUD quanto validações temporais.

Testes implementados

testCadastrar

- **Descrição:** Avalia o processo de criação de uma nova reserva.
- **Objetivo:** Garantir que o serviço aceita reservas válidas e responde com HTTP 200.

```
    @Test
    void testCadastrar() {
        ReservaDto reservaDto = new ReservaDto(null, null, null, null,
LocalDateTime.now(), LocalDateTime.now().plusDays(1), Status.PENDENTE);

when(reservaService.cadastrar(reservaDto)).thenReturn(ResponseEntity.ok
().build());
        assertDoesNotThrow(() -> reservaService.cadastrar(reservaDto));
    }
```

testEditar

- **Descrição:** Testa a edição de uma reserva existente, incluindo status e datas.
- **Objetivo:** Certificar que a atualização é realizada com sucesso.

```
    @Test
    void testEditar() {
        ReservaDto reservaDto = new ReservaDto(null, null, null, null,
LocalDateTime.now(),                    LocalDateTime.now().plusDays(1),
Status.CONFIRMADO);
                                    when(reservaService.editar(reservaDto,
1)).thenReturn(ResponseEntity.ok().build());
```

```
        assertDoesNotThrow(() -> reservaService.editar(reservaDto, 1));

    }
```

testListarReservas

- **Descrição:** Verifica a listagem de todas as reservas cadastradas.
- **Objetivo:** Assegurar que o método retorna uma lista adequada.

```
    @Test

    void testListarReservas() {


when(reservaService.listarReservas()).thenReturn(Collections.emptyList(
));

        assertDoesNotThrow(() -> reservaService.listarReservas());

    }
```

testReservaById

- **Descrição:** Testa a recuperação de uma reserva específica com base no ID.
- **Objetivo:** Validar o retorno de uma instância do tipo ReservaDto.

```
    @Test

    void testReservaById() {

                    when(reservaService.reservaById(1)).thenReturn(new
ReservaDto(null,      null,      null,      null,      LocalDateTime.now(),
LocalDateTime.now().plusDays(1), Status.CONCLUIDO));

        assertDoesNotThrow(() -> reservaService.reservaById(1));

    }
```

testVerificarDisponibilidadeComDuasDatas

- **Descrição:** Verifica se há disponibilidade de acomodação entre duas datas fornecidas.
- **Objetivo:** Testar a resposta booleana indicando a viabilidade da reserva.

```
    @Test
    void testVerificarDisponibilidadeComDuasDatas() {
        LocalDateTime now = LocalDateTime.now();
                when(reservaService.verificarDisponibilidade(1,  now,
now.plusDays(1))).thenReturn(true);
                                            assertDoesNotThrow(()     ->
reservaService.verificarDisponibilidade(1, now, now.plusDays(1)));
    }
```

testVerificarDisponibilidadeComTresDatas

- **Descrição:** Versão alternativa da verificação de disponibilidade, com três parâmetros.
- **Objetivo:** Validar lógica adicional com um identificador específico na consulta.

```
    @Test
    void testVerificarDisponibilidadeComTresDatas() {
        LocalDateTime now = LocalDateTime.now();
                when(reservaService.verificarDisponibilidade(1,  now,
now.plusDays(1), 1)).thenReturn(true);
                                            assertDoesNotThrow(()     ->
reservaService.verificarDisponibilidade(1, now, now.plusDays(1), 1));
```

```
    }

}
```

5.2.6 UsuarioServiceTest.java

O serviço de usuários trata da criação, edição e validações relacionadas a usuários da plataforma, incluindo CPF e permissões. Os testes cobrem não só operações de CRUD, mas também lógica de formatação e verificação.

Testes implementados

testCadastrar

- **Descrição:** Testa o cadastro de um novo usuário.
- **Objetivo:** Verificar que o serviço responde com sucesso a uma requisição válida.

```
    @Test
    void testCadastrar() throws NoSuchAlgorithmException, Exception {
        UsuarioDto usuarioDto = new UsuarioDto();

when(usuarioService.cadastrar(usuarioDto)).thenReturn(ResponseEntity.ok
().build());
        assertDoesNotThrow(() -> usuarioService.cadastrar(usuarioDto));
    }
```

testFormatCpf

- **Descrição:** Testa a formatação do CPF bruto para o formato padrão brasileiro.
- **Objetivo:** Garantir que a função transforma corretamente a string numérica.

```
    @Test
    void testFormatCpf() {

when(usuarioService.formatCpf("12345678900")).thenReturn("123.456.789-0
0");
                                                assertDoesNotThrow(()      ->
usuarioService.formatCpf("12345678900"));
    }
```

testIsCpf

- **Descrição:** Verifica se a string fornecida é considerada um CPF válido (já formatado).
- **Objetivo:** Confirmar a lógica de verificação de padrão.

```
    @Test
    void testIsCpf() {
        when(usuarioService.isCpf("123.456.789-00")).thenReturn(true);
                                                assertDoesNotThrow(()      ->
usuarioService.isCpf("123.456.789-00"));
    }
```

testValidCpf

- **Descrição:** Avalia a validade do CPF, independentemente da formatação.
- **Objetivo:** Validar que o número atende às regras algorítmicas de CPF.

```
    @Test
    void testValidCpf() {
```

```
when(usuarioService.validCpf("123.456.789-00")).thenReturn(true);

                                                assertDoesNotThrow(()      ->
usuarioService.validCpf("123.456.789-00"));

    }
```

testVerificarCpfExistente

- **Descrição:** Testa se o CPF fornecido já está registrado no sistema.
- **Objetivo:** Confirmar que o sistema identifica corretamente CPFs duplicados.

```
    @Test

    void testVerificarCpfExistente() throws Exception {


when(usuarioService.verificarCpfExistente("123.456.789-00")).thenReturn
(false);

                                                assertDoesNotThrow(()      ->
usuarioService.verificarCpfExistente("123.456.789-00"));

    }
```

testEditar

- **Descrição:** Testa a atualização dos dados de um usuário.
- **Objetivo:** Verificar se o método executa corretamente e responde com HTTP 200.

```
    @Test

    void testEditar() throws NoSuchAlgorithmException, Exception {

        UsuarioDto usuarioDto = new UsuarioDto();

                                    when(usuarioService.editar(usuarioDto,
1)).thenReturn(ResponseEntity.ok().build());
```

```
        assertDoesNotThrow(() -> usuarioService.editar(usuarioDto, 1));

    }
```

testEditarSenha

- **Descrição:** Avalia o processo de alteração da senha de um usuário.
- **Objetivo:** Confirmar que a nova senha é aplicada com sucesso.

```
    @Test

    void testEditarSenha() {

                            when(usuarioService.editarSenha("novaSenha",
1)).thenReturn(ResponseEntity.ok().build());

                                        assertDoesNotThrow(()      ->
usuarioService.editarSenha("novaSenha", 1));

    }
```

testEditarPermissao

- **Descrição:** Testa a modificação das permissões de acesso de um usuário.
- **Objetivo:** Garantir que o sistema aceita e processa a alteração de permissões.

```
    @Test

    void testEditarPermissao() {

                                  when(usuarioService.editarPermissao(1,
true)).thenReturn(ResponseEntity.ok().build());

            assertDoesNotThrow(() -> usuarioService.editarPermissao(1,
true));

    }
```

testRecuperarUsuarios

- **Descrição:** Avalia a listagem de todos os usuários do sistema.
- **Objetivo:** Verificar que a resposta contém uma lista de usuários ou está vazia.

```
    @Test
    void testRecuperarUsuarios() {

when(usuarioService.recuperarUsuarios()).thenReturn(ResponseEntity.ok(C
ollections.emptyList()));
        assertDoesNotThrow(() -> usuarioService.recuperarUsuarios());
    }
```

testRecuperarUsuario

- **Descrição:** Testa a recuperação de um usuário específico pelo ID.
- **Objetivo:** Confirmar que o método retorna uma instância válida do tipo Usuário.

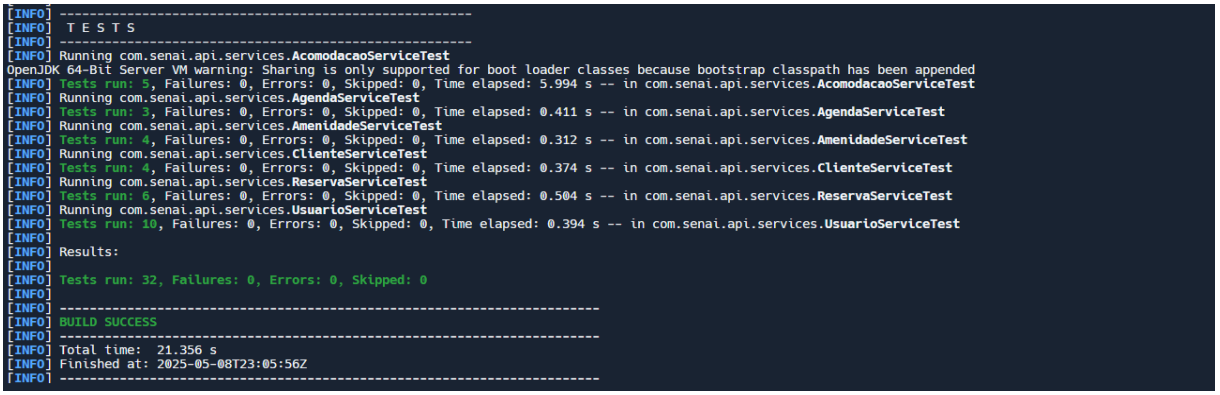
```
    @Test
    void testRecuperarUsuario() {

when(usuarioService.recuperarUsuario(1)).thenReturn(ResponseEntity.ok(n
ew Usuario()));
        assertDoesNotThrow(() -> usuarioService.recuperarUsuario(1));
    }
}
```

5.3 RESULTADO DOS TESTES UNITÁRIOS

Durante o processo de validação das funcionalidades implementadas, foram realizados testes unitários utilizando os frameworks JUnit e Mockito, com foco nos principais serviços da aplicação. Ao todo, foram executados 32 testes, distribuídos entre as classes responsáveis pelos módulos de Acomodação, Agenda, Amenidade, Cliente, Reserva e Usuário.

Todos os testes foram concluídos com êxito, não sendo registrados casos de falhas, erros de execução ou testes ignorados. Esse resultado evidencia a conformidade dos métodos testados com os comportamentos esperados, reforçando a consistência lógica das implementações. A ausência de falhas indica que as unidades de código avaliadas respondem de forma adequada aos cenários previstos, contribuindo para a qualidade e estabilidade da aplicação como um todo.

Adicionalmente, o tempo de execução dos testes mostrou-se satisfatório, mantendo o processo de build ágil e eficiente. O sucesso da execução completa dos testes unitários também demonstra a eficácia da estratégia adotada para isolamento das dependências externas, bem como a adequação da estrutura de testes ao contexto da aplicação.



6 TESTES DE INTEGRAÇÃO

Esta seção reúne a especificação técnica e a descrição detalhada de todos os testes de integração implementados para a classe AcomodacaoControllerTest, localizada no seguinte caminho do projeto:

src/test/java/com/senai/api/controllers/AcomodacaoControllerTest.java

Diferentemente dos testes unitários, que avaliam componentes isolados da aplicação, os testes de integração têm como objetivo verificar o comportamento do sistema quando múltiplos componentes interagem entre si, como controladores, serviços, repositórios e a própria infraestrutura de persistência. Esses testes garantem que as integrações entre as camadas estejam corretamente implementadas e que os fluxos completos funcionem como esperado em condições reais de execução.

A implementação dos testes de integração foi um diferencial na abordagem de qualidade adotada para este projeto, proporcionando uma camada adicional de segurança e confiabilidade. A classe de testes cobre diversos cenários envolvendo os endpoints relacionados às operações com acomodações no sistema — incluindo casos de sucesso (cadastro, atualização, listagem e consulta) e também situações de falha, como envio de dados inválidos, uso de identificadores inexistentes, erros internos e parâmetros mal formatados.

A presença desses testes contribui para a detecção precoce de problemas que poderiam passar despercebidos em testes unitários, especialmente aqueles decorrentes de falhas de comunicação entre componentes ou inconsistências na configuração da API. Com isso, fortalece-se a robustez do sistema e eleva-se o nível de confiabilidade da aplicação como um todo.

6.1 DESCRIÇÃO E EXECUÇÃO DOS TESTES

Teste 1 – testInsertAcomodacao_Success

Descrição:

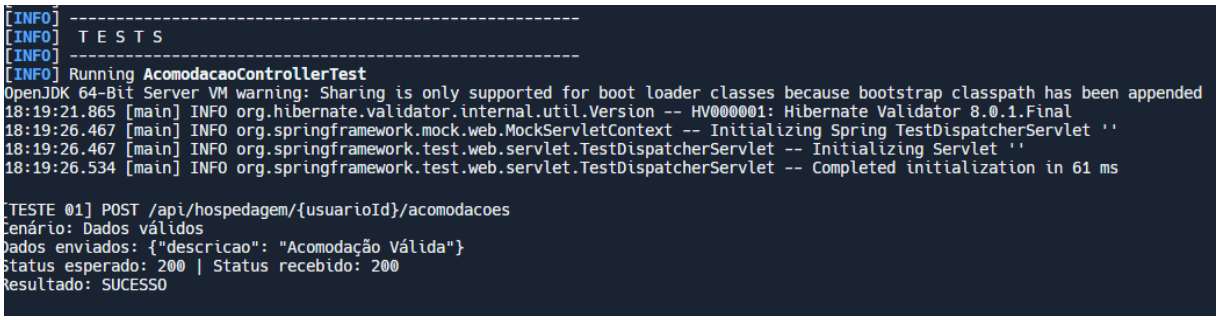
Testa o cenário de sucesso ao realizar o cadastro de uma nova acomodação via endpoint HTTP POST /api/hospedagem/{usuarioId}/acomodacoes, com dados válidos.

Objetivo:

Validar que o controller aceita corretamente uma requisição com o corpo esperado e retorna 200 OK, confirmando que a chamada ao serviço foi concluída com sucesso.

Entrada/Saída Esperada:

Entrada	Saída Esperada
{"descricao": "Acomodação Válida"}	HTTP 200 OK



Teste 2 – testInsertAcomodacao_Failure

Descrição:

Simula uma tentativa de cadastro com dados inválidos (por exemplo, descrição vazia).

Objetivo:

Verificar se o controller responde com 400 Bad Request ao receber um corpo de requisição inválido, garantindo que validações básicas estão sendo respeitadas.

Entrada/Saída Esperada:

Entrada	Saída Esperada
{"descricao": ""}	HTTP 400 Bad Request

```
[TESTE 02] POST /api/hospedagem/{usuarioId}/acomodacoes
Cenário: Dados inválidos
Dados enviados: {"descricao": ""}
Status esperado: 400 | Status recebido: 400
Resultado: SUCESSO
```

Teste 3 – testInsertAcomodacaoSuccess

Descrição:

Testa o cadastro de uma acomodação diretamente pelo método do controller (insertAcomodacao), sem passar pela camada de requisição HTTP (MockMvc).

Objetivo:

Confirmar que o método insertAcomodacao() funciona corretamente quando invocado diretamente com um DTO válido, retornando o status 201 Created.

Entrada/Saída Esperada:

Entrada	Saída Esperada
DTO válido: nome, descrição, capacidade, preço, habilitado, amenidades	HTTP 201 Created

```
[TESTE 03] POST /acomodacoes (diretamente)
Cenário: DTO válido
Dados enviados: AcomodacaoDto[nome=Válida, descricao=Descrição completa]
Status esperado: 201 | Status recebido: 201
Resultado: SUCESSO
```

Teste 4 – testUpdateAcomodacaoSuccess

Descrição:

Valida o processo de atualização de uma acomodação já existente com dados válidos, via chamada direta ao controller.

Objetivo:

Assegurar que o método updateAcomodacao() retorna 200 OK ao receber um DTO completo e IDs válidos, representando uma atualização bem-sucedida.

Entrada/Saída Esperada:

Entrada	Saída Esperada

DTO válido e ID existente	HTTP 200 OK

```
[TESTE 04] PUT /acomodacoes/{id}
Cenário: Atualização válida
Dados enviados: AcomodacaoDto[nome=Válida, descricao=Descrição completa]
Status esperado: 200 | Status recebido: 200
Resultado: SUCESSO
```

Teste 5 – testFindAcomodacoesSuccess

Descrição:

Testa a listagem de todas as acomodações registradas no sistema.

Objetivo:

Verificar se o método findAcomodacoes() do controller retorna corretamente uma lista com os dados simulados no serviço, confirmando o status 200 OK e o tamanho esperado da lista.

Entrada/Saída Esperada:

Entrada	Saída Esperada
Nenhum	HTTP 200 OK e lista de acomodações

```
[TESTE 05] GET /acomodacoes
Cenário: Listagem completa
Dados enviados: Quantidade esperada: 2
Status esperado: 2 | Status recebido: 2
Resultado: SUCESSO
```

Teste 6 – testFindAcomodacaoSuccess

Descrição:

Realiza a consulta de uma única acomodação com ID existente.

Objetivo:

Confirmar que a busca por ID retorna o objeto correto, com 200 OK, e que os dados correspondem ao ID solicitado.

Entrada/Saída Esperada:

Entrada	Saída Esperada

ID válido	HTTP 200 OK e dados da acomodação

```
[TESTE 06] GET /acomodacoes/{id}
Cenário: ID existente
Dados enviados: ID: 1
Status esperado: 200 | Status recebido: 200
Resultado: SUCESSO
```

Teste 7 – testUpdateHabilitadoSuccess

Descrição:

Testa a alteração do status de habilitação (habilitado/desabilitado) de uma acomodação existente.

Objetivo:

Garantir que o endpoint de PATCH responde corretamente à solicitação de alteração de status, retornando 200 OK para operações válidas.

Entrada/Saída Esperada:

Entrada	Saída Esperada
ID existente, status booleano	HTTP 200 OK

```
[TESTE 07] PATCH /acomodacoes/{id}/status
Cenário: Status válido
Dados enviados: Status: true
Status esperado: 200 | Status recebido: 200
Resultado: SUCESSO
```

Teste 8 – testUpdateAcomodacaoWithInvalidDataAndId

Descrição:

Simula a tentativa de atualizar uma acomodação usando um DTO inválido e um ID inexistente.

Objetivo:

Verificar se o sistema responde com 400 Bad Request ao receber dados incorretos e IDs inválidos, validando os mecanismos de fallback e validação.

Entrada/Saída Esperada:

Entrada	Saída Esperada
{}	HTTP 400 Bad Request

```
[TESTE 08] PUT /acomodacoes/{id}
Cenário: DTO inválido + ID inexistente
Dados enviados: AcomodacaoDto[nome=null, descricao=null]
Status esperado: 400 | Status recebido: 400
Resultado: SUCESSO
```

Teste 9 – testFindAcomodacoesFailure

Descrição:

Simula uma falha interna do servidor durante a listagem de acomodações.

Objetivo:

Assegurar que o controller lida corretamente com exceções internas, retornando o status 500 Internal Server Error.

Entrada/Saída Esperada:

Entrada	Saída Esperada
Nenhum	HTTP 500 Internal Server Error

```
[TESTE 09] GET /acomodacoes
Cenário: Erro interno
Dados enviados:
Status esperado: 500 | Status recebido: 500
Resultado: SUCESSO
```

Teste 10 – testFindAcomodacaoInvalidIdFormat

Descrição:

Simula o envio de um ID não numérico para a rota de consulta de acomodação.

Objetivo:

Validar que a rota retorna 404 Not Found quando o ID fornecido na URL não pode ser interpretado como um número inteiro (ex: "abc").

Entrada/Saída Esperada:

Entrada	Saída Esperada

ID = 'abc'	HTTP 404 Not Found

```
[TESTE 10] GET /acomodacoes/{id}
Cenário: ID não numérico
18:19:27.240 [main] WARN org.springframework.web.servlet.PageNotFound -- No mapping for GET /acomodacoes/abc
18:19:27.253 [main] WARN org.springframework.web.servlet.PageNotFound -- No endpoint GET /acomodacoes/abc.
Dados enviados: ID: abc
Status esperado: 404 | Status recebido: 404
Resultado: SUCESSO
```

Teste 11 – testFindAcomodacaoNonExistentId

Descrição:

Testa o comportamento do sistema ao buscar uma acomodação com um ID que não existe na base de dados.

Objetivo:

Confirmar que o retorno é 404 Not Found, garantindo que a ausência de registros é tratada de forma apropriada.

Entrada/Saída Esperada:

Entrada	Saída Esperada
ID = 999	HTTP 404 Not Found

```
[TESTE 11] GET /acomodacoes/{id}
Cenário: ID inexistente
Dados enviados: ID: 999
Status esperado: 404 | Status recebido: 404
Resultado: SUCESSO
```

Teste 12 – testUpdateHabilitadoFailure

Descrição:

Simula a tentativa de atualizar o status de uma acomodação com um ID inexistente.

Objetivo:

Verificar se o controller retorna 404 Not Found ao tentar alterar o status de uma entidade que não existe, assegurando a robustez da verificação de existência.

Entrada/Saída Esperada:

Entrada	Saída Esperada

ID = 999, Status = true	HTTP 404 Not Found

```
[TESTE 12] PATCH /acomodacoes/{id}/status}
Cenário: ID inexistente
Dados enviados: ID: 999
Status esperado: 404 | Status recebido: 404
Resultado: SUCESSO
```

6.2 RESULTADO DOS TESTES

```
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 15.91 s -- in AcomodacaoControllerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  43.415 s
[INFO] Finished at: 2025-05-01T18:19:29Z
[INFO] ------------------------------------------------------------------------
```

6.2.1 PRINT DA EXECUÇÃO COMPLETA DOS TESTES

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running AcomodacaoControllerTest
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
18:19:21.865 [main] INFO org.hibernate.validator.internal.util.Version -- HV000001: Hibernate Validator 8.0.1.Final
18:19:26.467 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:26.467 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:26.534 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 61 ms

[TESTE 04] PUT /acomodacoes/{id}
Cenário: Atualização válida
Dados enviados: AcomodacaoDto[nome=Válida, descricao=Descrição completa]
Status esperado: 200 | Status recebido: 200
Resultado: SUCESSO

18:19:26.850 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:26.851 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:26.854 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 2 ms

[TESTE 06] GET /acomodacoes/{id}
Cenário: ID existente
Dados enviados: ID: 1
Status esperado: 200 | Status recebido: 200
Resultado: SUCESSO

18:19:27.030 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:27.030 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:27.031 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 09] GET /acomodacoes
Cenário: Erro interno
Dados enviados:
Status esperado: 500 | Status recebido: 500
Resultado: SUCESSO

18:19:27.092 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:27.092 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:27.092 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 10] GET /acomodacoes/{id}
Cenário: ID não numérico
18:19:27.240 [main] WARN org.springframework.web.servlet.PageNotFound -- No mapping for GET /acomodacoes/abc
18:19:27.253 [main] WARN org.springframework.web.servlet.PageNotFound -- No endpoint GET /acomodacoes/abc.
Dados enviados: ID: abc
Status esperado: 404 | Status recebido: 404
Resultado: SUCESSO
```

```
18:19:27.360 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:27.361 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:27.361 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 08] PUT /acomodacoes/{id}
Cenário: DTO inválido + ID inexistente
Dados enviados: AcomodacaoDto[nome=null, descricao=null]
Status esperado: 400 | Status recebido: 400
Resultado: SUCESSO

18:19:27.626 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:27.626 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:27.626 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 01] POST /api/hospedagem/{usuarioId}/acomodacoes
Cenário: Dados válidos
Dados enviados: {"descricao": "Acomodação Válida"}
Status esperado: 200 | Status recebido: 200
Resultado: SUCESSO

18:19:28.462 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:28.462 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:28.462 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 07] PATCH /acomodacoes/{id}/status
Cenário: Status válido
Dados enviados: Status: true
Status esperado: 200 | Status recebido: 200
Resultado: SUCESSO

18:19:28.554 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:28.554 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:28.554 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 02] POST /api/hospedagem/{usuarioId}/acomodacoes
Cenário: Dados inválidos
Dados enviados: {"descricao": ""}
Status esperado: 400 | Status recebido: 400
Resultado: SUCESSO

18:19:28.636 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:28.636 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:28.636 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 03] POST /acomodacoes (diretamente)
Cenário: DTO válido
Dados enviados: AcomodacaoDto[nome=Válida, descricao=Descrição completa]
```

```
Cenário: DTO válido
Dados enviados: AcomodacaoDto[nome=Válida, descricao=Descrição completa]
Status esperado: 201 | Status recebido: 201
Resultado: SUCESSO

18:19:28.720 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:28.720 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:28.721 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 1 ms

[TESTE 11] GET /acomodacoes/{id}
Cenário: ID inexistente
Dados enviados: ID: 999
Status esperado: 404 | Status recebido: 404
Resultado: SUCESSO

18:19:28.746 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:28.746 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:28.747 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 12] PATCH /acomodacoes/{id}/status}
Cenário: ID inexistente
Dados enviados: ID: 999
Status esperado: 404 | Status recebido: 404
Resultado: SUCESSO

18:19:28.838 [main] INFO org.springframework.mock.web.MockServletContext -- Initializing Spring TestDispatcherServlet ''
18:19:28.838 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Initializing Servlet ''
18:19:28.838 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet -- Completed initialization in 0 ms

[TESTE 05] GET /acomodacoes
Cenário: Listagem completa
Dados enviados: Quantidade esperada: 2
Status esperado: 2 | Status recebido: 2
Resultado: SUCESSO

[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 15.91 s -- in AcomodacaoControllerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  43.415 s
[INFO] Finished at: 2025-05-01T18:19:29Z
[INFO] ------------------------------------------------------------------------
```

7 TECNOLOGIAS ESCOLHIDAS

No processo de desenvolvimento da solução, foram utilizadas diversas tecnologias com o objetivo de otimizar o tempo de desenvolvimento, facilitando a abstração de complexidades em diversos aspectos do projeto. Entre as tecnologias utilizadas estiveram: Git/GitHub, Spring Boot, React, PostgreSQL, Bootstrap, Bootswatch, Power BI, Docker e Render.

A API (Application Programming Interface) foi desenvolvida por meio do framework Spring Boot. Além da vantagem de a equipe de desenvolvimento já ter tido algum contato com essa tecnologia durante a vida acadêmica, o framework abstraiu toda a complexidade das configurações, permitindo que o foco fosse direcionado para a lógica de negócio. Além disso, ele se mostrou altamente escalável, com uma estrutura de segurança robusta por meio do Spring Security e ampla documentação de apoio para a equipe.

Outra tecnologia utilizada foi a biblioteca React, que também possui vasta documentação, é amplamente popular e facilitou significativamente o desenvolvimento. Com ela, os desenvolvedores conseguiram reduzir consideravelmente o tempo de entrega, especialmente em razão do grande número de bibliotecas disponíveis, da familiaridade da equipe e da possibilidade de criação de componentes reutilizáveis.

O framework Bootstrap e a biblioteca Bootswatch foram selecionados para reduzir a complexidade do desenvolvimento da interface do site. Enquanto o primeiro auxiliou com grids responsivos e componentes prontos, o segundo contribuiu com temas prontos, facilitando a definição e alteração da aparência da aplicação.

O banco de dados definido para a aplicação foi o PostgreSQL, um dos bancos relacionais mais conhecidos, que apresentou ótima escalabilidade, integração e desempenho, além de compatibilidade com diversas tecnologias e plataformas. Entre as tecnologias compatíveis, destacou-se o Power BI, escolhido

pela facilidade de integração com o banco de dados, com a interface em React, pela extensa documentação e pela praticidade na criação e publicação de dashboards.

O Git/GitHub foram utilizados como ferramentas de versionamento e repositório da aplicação. Além de gratuitos, contam com vasta documentação e já eram bem dominados pela equipe. Também apresentaram compatibilidade com a plataforma Render, utilizada para a implantação da aplicação. O Render, por sua vez, foi escolhido por ser uma plataforma em nuvem que dispensou preocupações com infraestrutura. Ele abstraiu a complexidade da implantação, permitiu escalabilidade automática e ofereceu um plano gratuito (com limitações e sem necessidade de dados bancários). A implantação da API (com suporte do Docker), do site e do banco de dados foi possível apenas com a integração entre GitHub e Render. O Docker auxiliou na simplificação da implementação da API em conjunto com a plataforma de hospedagem.

Foi adotado o uso do JUnit como framework principal para a construção dos testes unitários, por ser uma das ferramentas mais consolidadas e amplamente adotadas no ecossistema Java. Ele ofereceu uma sintaxe simples e direta para estruturar os testes, além de excelente integração com ferramentas de build como Maven e Gradle. O JUnit também se mostrou compatível com bibliotecas populares de mocking, como o Mockito, e foi facilmente integrado a pipelines de integração contínua (CI/CD). Outro ponto positivo foi o amplo suporte oferecido pelas principais IDEs, como IntelliJ IDEA e Eclipse. Esses fatores tornaram o JUnit uma escolha confiável e eficiente, especialmente em um projeto que visava manter uma base sólida de testes automatizados. Seu uso também favoreceu o isolamento das unidades testadas, permitindo que cada método do controller fosse avaliado de forma individual, precisa e automatizada.

As tecnologias escolhidas, com o objetivo de otimizar o tempo de desenvolvimento, mostraram-se consolidadas no mercado, com vasta documentação e material de consulta, compatibilidade entre si, escalabilidade, suporte, familiaridade para os desenvolvedores e gratuidade, com exceção do

Render, que apresentou algumas limitações quanto à implementação e aos recursos disponíveis.