FIAP - 3ESPY - 2025 - Profa. Damiana Costa

Arquitetura Orientado a Serviço

CP 2 - Semana 10 - 20/10/2025

📌 Trabalho: Sistema de Reserva de Hotel (Check-in e Check-out)

> **Dica** — Enunciado agnóstico à linguagem. Você pode implementar em **Java, C#, Node.js, Python, Go, etc.** O foco é **na arquitetura, regras de negócio e boas práticas**, não na stack específica.

---

1) Objetivo

Desenvolver uma **API REST** para gestão de reservas de hotel, cobrindo o ciclo de vida **reserva → check-in → check-out**, seguindo **arquitetura em 3 camadas (MVC)** e boas práticas de **Status Code/Verbos HTTP**, com persistência por qualquer tecnologia desde que haja **migração versionada de banco** e **validação de dados**. O projeto deve incluir **tratamento de exceções** e um **README** completo para execução e avaliação.

Ao final, espera-se que a aplicação:

- **Cadastre hóspedes e quartos**, crie/atualize/cancele **reservas**, e **registre check-in e check-out**;
- **Valide regras de negócio** (disponibilidade, datas, estado da reserva, capacidade, etc.);
- Responda com **Status Codes adequados** e payloads claros (**DTOs**);
- Seja **executável localmente** (H2/MySQL/PostgreSQL, conforme README), com **migrações** e **seeds**.

---

2) Escopo funcional (MVP)

Entidades principais

- **Hóspede** (Guest): nome completo, documento (CPF/passaporte), e-mail, telefone.
- **Quarto** (Room): número (único), tipo (STANDARD | DELUXE | SUITE), capacidade (nº hóspedes), preço base por diária, status (ATIVO/INATIVO).
- **Reserva** (Reservation): hóspede, quarto, **dataCheckInPrevisto**, **dataCheckOutPrevisto**, status (CREATED | CHECKED_IN | CHECKED_OUT | CANCELED), valorEstimado e valorFinal (opcional), timestamps (criação/atualização).

Fluxos obrigatórios

1. **Cadastro e consulta** de Hóspedes e Quartos (CRUD básico).
2. **Criar Reserva**: valida disponibilidade do quarto no período, regra de datas e capacidade.
3. **Check-in**: altera status para `CHECKED_IN`, valida política de janela de check-in (ver regras).
4. **Check-out**: altera status para `CHECKED_OUT`, calcula valor final (baseado em diárias).
5. **Cancelar Reserva**: permitido somente enquanto `CREATED` (antes do check-in).

---

3) Regras de negócio (validar na camada de domínio/serviço)

1. **Datas válidas**: `dataCheckOutPrevisto` **>** `dataCheckInPrevisto`.
   - Violação → **400 Bad Request** + `InvalidDateRangeException` (ou equivalente).
2. **Disponibilidade do quarto**: um quarto **não pode** ter reservas com **sobreposição** no período solicitado, exceto reservas canceladas.
   - Violação → **409 Conflict** + `RoomUnavailableException`.
3. **Capacidade**: nº de hóspedes **≤ capacidade** do quarto.
   - Violação → **400 Bad Request** + `CapacityExceededException`.
4. **Transições de status (FSM)**:
   - `CREATED → CHECKED_IN → CHECKED_OUT`
   - `CREATED → CANCELED`
   - Violação (ex.: check-out sem check-in, cancelar após check-in) → **409 Conflict** + `InvalidReservationStateException`.
5. **Janela de check-in**: permitir check-in **no dia de** `dataCheckInPrevisto` (ou política alternativa **documentada** no README).
   - Fora da política → **422 Unprocessable Entity** (opcional).
6. **Cálculo do valor (check-out)**:
   - `diarias = max(1, diasEntre(checkinEfetivo, checkoutEfetivo))`
   - `valorFinal = diarias * precoBaseQuarto`
   - (Opcional) `valorEstimado` = diárias previstas * preço base.
7. **Remoção de entidades**: **não** excluir fisicamente quartos com reservas; usar **INATIVO** ou bloqueio.
   - Violação → **409 Conflict**.

---

4) Requisitos técnicos (agnósticos a linguagem)

- **Arquitetura**: 3 camadas (**Controller → Service/Domain → Repository/DAO**), com **DTOs** para entrada/saída.
- **API REST**: recursos, verbos e status adequados;
- **Persistência**: escolha livre (ORM ou SQL/JDBC/Client nativo), mas:
  - **Migrações versionadas** (ex.: **Flyway/Liquibase/Prisma Migrate/Knex Migrate**);
  - **Seeds** para dados iniciais.
- **Validação de dados**: usar o mecanismo da linguagem (ex.: **Jakarta Bean Validation**, **FluentValidation**, **class-validator**, **Pydantic**, etc.). **Validar nos DTOs de entrada**.
- **Tratamento de exceções**.
- **Documentação**: **README** com instruções de execução, DB, migrações, endpoints e exemplos (cURL/insomnia/postman).
- **Contrato de API**: **OpenAPI/Swagger** (YAML/JSON) **ou** coleção Postman/Insomnia exportada.

---

5) Modelo de dados (SQL portável — H2 / MySQL / PostgreSQL)

> **Observação:** IDs como `CHAR(36)` (UUID em string) para máxima portabilidade. Caso use tipos nativos de UUID (PostgreSQL), adapte o script.

```
-- ================================================================
-- RESET (ordem de FKs) — ajuste para seu banco/ambiente
-- ================================================================
```

```
DROP DATABASE IF EXISTS sishotel;
CREATE DATABASE sishotel;

USE sishotel;

DROP TABLE IF EXISTS reservations;
DROP TABLE IF EXISTS rooms;
DROP TABLE IF EXISTS guests;

-- ================================================================
-- GUESTS
-- ================================================================
CREATE TABLE guests (
  id               CHAR(36)     NOT NULL,
  full_name        VARCHAR(120) NOT NULL,
  document         VARCHAR(30)  NOT NULL,
  email            VARCHAR(120) NOT NULL,
  phone            VARCHAR(30),
  created_at       TIMESTAMP    DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_guests PRIMARY KEY (id),
  CONSTRAINT uq_guests_document UNIQUE (document),
  CONSTRAINT uq_guests_email UNIQUE (email)
);

-- ================================================================
-- ROOMS
-- ================================================================
CREATE TABLE rooms (
  id                CHAR(36)      NOT NULL,
  number            INT           NOT NULL,
  type              VARCHAR(20)   NOT NULL, -- STANDARD | DELUXE | SUITE
  capacity          INT           NOT NULL,
  price_per_night   DECIMAL(10,2) NOT NULL,
  status            VARCHAR(20)   NOT NULL, -- ATIVO | INATIVO
  CONSTRAINT pk_rooms PRIMARY KEY (id),
  CONSTRAINT uq_rooms_number UNIQUE (number)
);

-- ================================================================
-- RESERVATIONS
-- ================================================================
CREATE TABLE reservations (
  id                  CHAR(36)     NOT NULL,
  guest_id            CHAR(36)     NOT NULL,
  room_id             CHAR(36)     NOT NULL,
  checkin_expected    DATE         NOT NULL,
  checkout_expected   DATE         NOT NULL,
  checkin_at          TIMESTAMP,
  checkout_at         TIMESTAMP,
  status              VARCHAR(20)  NOT NULL, -- CREATED | CHECKED_IN |
CHECKED_OUT | CANCELED
  estimated_amount    DECIMAL(10,2),
  final_amount        DECIMAL(10,2),
  created_at          TIMESTAMP    DEFAULT CURRENT_TIMESTAMP,
```

```
  updated_at          TIMESTAMP,
  CONSTRAINT pk_reservations PRIMARY KEY (id),
  CONSTRAINT fk_reservations_guest FOREIGN KEY (guest_id) REFERENCES
guests(id),
  CONSTRAINT fk_reservations_room  FOREIGN KEY (room_id)  REFERENCES
rooms(id)
);

-- Índices úteis
CREATE INDEX idx_rooms_status ON rooms (status);
CREATE INDEX idx_reservations_room ON reservations (room_id);
CREATE INDEX idx_reservations_status ON reservations (status);
CREATE INDEX idx_reservations_date_range ON reservations
(checkin_expected, checkout_expected);

-- ================================================================
-- SEED (exemplos) — substitua IDs por UUIDs gerados pela sua aplicação,
se preferir
-- ================================================================
INSERT INTO guests (id, full_name, document, email, phone)
VALUES
('11111111-1111-1111-1111-111111111111', 'Ana Silva', '12345678901',
'ana@example.com', '+55-11-99999-1111'),
('22222222-2222-2222-2222-222222222222', 'Bruno Souza', '98765432100',
'bruno@example.com', '+55-21-98888-2222');

INSERT INTO rooms (id, number, type, capacity, price_per_night, status)
VALUES
('aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa', 101, 'STANDARD', 2, 250.00,
'ATIVO'),
('bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', 201, 'DELUXE', 3, 380.00,
'ATIVO'),
('cccccccc-cccc-cccc-cccc-cccccccccccc', 301, 'SUITE', 4, 520.00,
'ATIVO');

INSERT INTO reservations (
  id, guest_id, room_id, checkin_expected, checkout_expected, status,
estimated_amount, created_at
) VALUES
('99999999-9999-9999-9999-999999999999',
 '11111111-1111-1111-1111-111111111111',
 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa',
 DATE '2025-11-05', DATE '2025-11-07', 'CREATED', 2 * 250.00,
CURRENT_TIMESTAMP
);

-- Notas de compatibilidade
-- H2: jdbc:h2:mem:hotel;MODE=PostgreSQL;DB_CLOSE_DELAY=-1
-- MySQL: se necessário, troque DATE '2025-11-05' por '2025-11-05'
-- PostgreSQL: funciona como está; se preferir, troque CHAR(36) por UUID
```

> **Regra de sobreposição (na aplicação):** conflito se `entradaA < saídaB` **e** `entradaB < saídaA` (mesmo quarto).

---

6) Critérios de avaliação (100 pts)

Critério	Pts
Arquitetura MVC bem definida	10
API REST (verbos e Status Codes corretos)	15
Persistência (DAO/Repository; migrações e seed)	12
Regras de negócio implementadas	15
Tratamento de Exceções	10
Contrato de API (Swagger/OpenAPI ou Collections Postman/Insomnia)	8
Swagger completo e funcional (se optar por Swagger)	5
README completo (execução, decisões, endpoints)	10
Validação de dados (DTOs)	10
Organização geral do projeto	5

---

8) Entregáveis (o que exatamente deve ser entregue)

1. **Código-fonte** em repositório Git (público ou privado com acesso).
2. **README.md** contendo obrigatoriamente:
   - Stack escolhida e **como executar** localmente;
   - Configuração do **banco**, **migrações** e **seeds**;
   - **Contratos de API** (Swagger/OpenAPI YAML/JSON **ou** coleção Postman/Insomnia exportada);
   - **Exemplos de requisições** (cURL/coleção) e respostas esperadas;
   - **Decisões de arquitetura** (2–3 ADRs curtas) e diagrama simples das camadas;
3. **Scripts de migração** versionados (ex.: `db/migration/V1__init.sql`) + **seed**.
4. **Tratamento de erros** com payload padronizado (código, mensagem, timestamp).

Formação de grupos: até **5 pessoas**.
Entrega: até **27/10/2025** via **Teams** (link do repositório + instruções).

---

9) Dicas finais

- Comece pela **modelagem** e regras de negócio; depois exponha via API.
- Trate **erros de negócio** como **409** (conflito) e **erros de campo** como **400**; **404** para inexistentes.
- Escreva **mensagens de erro** claras e consistentes.
- Se optar por Swagger/OpenAPI, gere o **arquivo estático** no repositório.
- Mantenha o escopo **enxuto**, porém **correto** — qualidade > quantidade.

Boa implementação! 🚀