Relátorio - Trabalho Design de Software

Leonardo Krambeck - GRR20190482
Erick Graeff Petzold - GRR20193175

30 de novembro de 2025

1 Justificativa da Arquitetura

Optou-se por arquitetura em camadas (API/Routers, Services, Domain, Repository) para separar responsabilidades e facilitar evolução. MVC puro, usado como API, tende a concentrar regras nos controllers e misturar protocolo HTTP com lógica de negócio. A divisão adotada isola contratos externos (rotas), regras e invariantes (services), linguagem de domínio (entidades, schemas, enums) e mecanismo de persistência (repositories), reduzindo impacto de mudanças.

Acreditamos que a opção de microserviços iria adicionar complexidade desnecessária ao código, pois é necessario que haja simplicidade na implementação. Apesar de ser bem escalável também aumenta overheads no desempenho enquanto a arquitetura em camadas favorece a simplicidade porém permitindo a divisão de responsabilidades, facilitando sua manutenção.

O banco de dados usamos um mock em JSON salvo localmente para não complicar muito a implementação do protótipo, o qual pode facilmente ser subtituido por um banco de dados PostgreSQL com a biblioteca SQLAlchemy do python.

Benefícios principais: testabilidade (services sem framework), facilidade de migração de persistência (JSON para banco relacional), clareza na aplicação das regras críticas (validação de período e prevenção de double-booking)

e trajetória simples para acrescentar autenticação ou cache. Evita complexidade prematura de microserviços ou DDD completo, mantendo estrutura enxuta e extensível.

2 Divisão do Código

O foco da implementação foi no desenvolvimento dos endpoints, visto que é onde se concentra as decisões da matéria estudada. Por conta disso optamos focar em aspectos mais relevantes da arquitetura do sistema, ou seja, na interação dos elementos e organização do código. Dessa forma não implementamos um front-end, apenas as APIs e abstração do banco de dados por arquivos JSON como descrita abaixo.

O código foi separado em camadas de acordo com o diagrama de componentes: camada de API, camada de aplicação/serviço, camada de dominio e camada de repositório. À seguir vamos descrever em mais detalhes cada uma delas.

2.1 API (Routers)

Responsável pelos endpoints REST e contratos externos. Cada recurso possui um roteador próprio em `src/app/routers/`, usando schemas de entrada/saída de `src/app/domain`. Não contém regras de negócio; apenas delega aos serviços e faz verificações simples dos campos.

2.2 Services

O serviço é o orquestrador do endpoint. Ele é responsável por fazer validações e requisições para cada entidade através dos Repositórios de acordo com as regras de negócio. Os serviços ficam em `src/app/service/` e coordenam chamadas a repositórios e composição de entidades.

2.3 Domain

Define a linguagem do domínio: entidades, value objects, regras puras e enums. Localizado em `src/app/domain/`. Mantém modelos independentes de protocolo HTTP e de persistência, facilitando testes e reuso.

2.4 Repository

Abstrai a persistência. Recebe as requisições dos services e fica responsável por traduzir para o banco de dados, servindo como uma interface. Inicialmente o banco de dados usa armazenamento em JSON local para facilitar a prototipação, portanto os repositórios implementam apenas o controle dos arquivos JSON através de funções do sistema operacional de controle de arquivos. Os respositórios ficam em `src/app/repository/`. Também concentra consultas específicas (por espaço e intervalo) necessárias para evitar conflitos.

2.5 Infraestrutura de DB

Configurações e modelos de acesso ao banco/mocks em `src/app/repository/db/`. No protótipo, inicializa arquivos e fornece utilitários de sessão; em versões futuras, gerencia conexão e migrações. Podende evoluir para Postgres/SQLAlchemy facilmente sem necesidade de alterar serviços ou rotas.