Seu Cantinho
Relatório - Design de Software

Augusto Antonio Kolb Schiavini GRR20232337
Cauê Mateus Gonçalves Venturin Samonek GRR20232339
Pedro Henrique Feijó Sampol GRR20232335

Outubro 2025

1 Introdução

Este relatório trata da elaboração de um protótipo de projeto, conforme a descrição do trabalho de Design de Software (Simone Dominico, Setembro 2025). O projeto se chama *Seu Cantinho*, e envolve uma rede de lojas de aluguel cuja proprietária é a Dona Maria, que precisa aperfeiçoar seu sistema devido ao rápido crescimento da empresa.

Para isso, foi necessário seguir os seguintes passos:

1. Elencar os *Requisitos Funcionais* e *Não Funcionais*;
2. Determinar quais requisitos não funcionais são *ASR*s (Requisitos Arquitetonicamente Significativos);
3. Propor uma arquitetura eficiente para o projeto, com diagramas de classe e de componentes;
4. Discorrer sobre o projeto e sua implementação.

Todos esses passos estão descritos a partir da Seção 2.

2 Arquitetura

2.1 Requisitos Funcionais

- RF01 - O Sistema deve cadastrar os espaços disponíveis para aluguel, com informações detalhadas como fotos, capacidade e preço.
- RF02 - O Sistema deve gerenciar as reservas futuras, contemplando data, cliente, espaço alugado e valor pago.
- RF03 - O Sistema deve controlar os pagamentos, indicando se a reserva possui apenas sinal pago ou se já está quitada.

2.2 Requisitos Não Funcionais

- RNF01 — Escalabilidade:O sistema deve suportar aumento de até 300% no volume de reservas sem perder mais de 20% de desempenho.
- RNF02 — Confiabilidade: Disponibilidade mínima de 99,5% e divergência entre filiais inferior a 0,01%.
- RNF03 — Usabilidade: Usuário deve conseguir fazer uma reserva em até 3 minutos, com taxa de erro menor que 5%.
- RNF04 — Simplicidade: Processo de reserva com no máximo 3 etapas e até 2 níveis de navegação.
- RNF05 — Manutenibilidade: Mudanças devem levar menos de 4 horas para serem aplicadas e exigir no máximo uma parada programada por mês.

2.3 Escolha da Arquitetura Alvo

Isto posto, nós entendemos que os requisitos RNF01 e RNF02 são *ASR*s. Nem todo modelo arquitetural garante tanta escalabilidade e robustez, sendo esta uma questão crucial para o projeto – enquanto os outros requisitos são mais periféricos, dependendo mais da implementação em si.

Nesse sentido, optamos por um modelo cliente-servidor orientado a eventos. Um modelo orientado a eventos permite que desacoplemos várias operações, de tal forma que podemos subir várias aplicações para executar as

mesmas tarefas, e com isso escalar o projeto e evitar problemas de falhas eventuais em algum processo. Algumas APIs serão responsáveis por colher os inputs do usuário e produzirão eventos solicitando a outros agentes o que foi pedido, como tentar alugar um certo local por exemplo. Enquanto outras APIs processarão essas requisições e aplicarão as regras de negócio, interagindo com um banco de dados para averiguar caso a caso e emitir eventos correspondentes. Ambas as APIs conectam-se num Broker, responsável por organizar a fila de eventos.

Dessa forma, os dados, os eventos referentes a eles e seu processamento estarão todos separados, assim, a falha em um campo não afetará diretamente outro.

2.4 Diagramação

Para este projeto, há diagramas de Classe e de Componentes tanto para a arquitetura ideal, explicada nas seções abaixo, quanto para uma implementação simplificada descrita em 3.5. Todos os diagramas se encontram em \uml.

3 Desenvolvimento e Implementação

3.1 Sobre o Domínio

3.1.1 Entidades

- Reserva (reservaId, clienteId, horário, imóvelId, status(confirmada, cancelada, pendente), invariantes: horários válidos e status possíveis)
- Cliente (clienteId, nome)
- Pagamento (reservaId, status(pago, sinal, pendente), meio, valor)
- Imóvel (imóvelId, capacidade, preço, localização)

3.1.2 Serviços de Domínio

- ServiçoDeDomínio(criarReserva, checarConflito, validarDisponibilidade, mudarStatusReserva); Responsabilidades: garante a regra anti-double-booking, aplica invariantes externas às entidades, coordena Reserva +

repositório, emite eventos de domínio (ReservaCriada, ReservaRecusada etc.)

- SistemaPagamento(registrarPagamento, validarPagamento); Responsabilidades: Confirmação da Reserva após quitar completamente o pagamento, eventos de domínio (PagamentoEfetuado)

3.1.3 Infraestrutura

- SistemaLogin(autenticar, gerarToken, validarSenha)
- Broker(gerenciarFilas, organizarEventos)
- GerenciadorDados(lerRegistro, alterarRegistro)
- Cache

3.2 Emissão de Eventos

No fluxo do sistema temos o seguinte: Quando o cliente solicita alguma funcionalidade que altera o estado do sistema, como alugar um espaço ou realizar o pagamento por exemplo, as APIs ServiçoDeImóveis e SistemaPagamento disparam eventos de acordo, que entram em suas respectivas filas de eventos do Broker – cada fila servindo para um tipo de evento, como solicitar aluguel, pagar dívida, resposta do pagamento da dívida etc.

Pensando em termos de escalonamento, podemos ter várias APIs ServiçoDeDomínio, que processam regras de negócio, escutando a fila. O Broker, tipicamente, é responsável por algumas tarefas: organizar os eventos, evitar duplicidade (impedir que o mesmo evento seja tratado por mais de uma API, distribuindo os eventos entre as APIs de forma organizada) e cuidar de DeadLetters (mensagens que por algum motivo não conseguiram ser enviadas). Com a gestão dos eventos funcionando, as APIs ServiçoDeDomínio recebem os eventos e processam a requisição. Para isso, elas interagem com o GerenciadorDados para checar certos registros e então validar algumas requisições ou até mesmo atualizar registros. Depois disso elas disparam eventos de respostas, que são incluídos em sua próprias filas de eventos no Broker e então devolvidos às APIs ServiçoDeImóveis.

3.3 Contrato e Conflito de Requisições

Somente o ServiçoDeDomínio pode alterar o estado de reservas; todas as operações de reserva devem consultar a disponibilidade via este serviço; nenhum outro componente pode modificar reservas diretamente. Suponhamos que dois clientes A e B tentam alugar o mesmo espaço ao mesmo tempo. Pois bem, o ServiçoDeImóveis de cara irá disparar eventos do tipo "alugar tal espaço"que entrarão na fila de eventos do Broker. A depender de como ache melhor, o Broker pode enviar um evento para uma API e outro para outra ou os dois para a mesma API. Quando o primeiro evento for processado e a requisição for validada no banco de dados, o registro será alterado para "alugado"e assim quando o processamento do segundo evento verificar o status do local, não conseguirá alugá-lo. Por fim, dois eventos (um de requisição aprovada e outro de preterida) serão disparadas e retornarão por meio do Broker ao ServiçoDeImóveis de cada cliente, sendo então responsável por tratar o caso para cada usuário. Essa dinâmica, que se manifesta como contrato arquitetural, depende da implementação do ServiçoDeDomínio, que fica responsável por se comunicar com o gerenciadorDados(mostra o status do imóvel) e determinar qual reserva poderá acontecer de fato – e assim ele dispara os respectivos eventos para cada cliente consumir (um dizendo que a reserva foi possível e outro que não).

3.4 Aperfeiçoamentos

Há também funcionalidades que não alteram o status do sistema, como checar se a dívida está paga ou olhar no catálogo quais as opções de imóveis. Nesses casos, pensamos que o SistemaPagamento e ServiçoDeImóveis podem interagir diretamente com o GerenciadorDados mediante componente de leitura (somente) para verificar as informações, sem precisar passar pelo Broker. Isso reduziria o número de eventos emitidos e o processamento das APIs ServiçoDeDomínio, tornando o processo mais rápido.

Evidentemente, o GerenciadorDados terá de lidar com muitas requisições, tanto de alterações quanto checagem. Nesse sentido pensamos em adicionar uma cache no GerenciadorDados para evitar que ele chame o banco de dados com muita frequência.

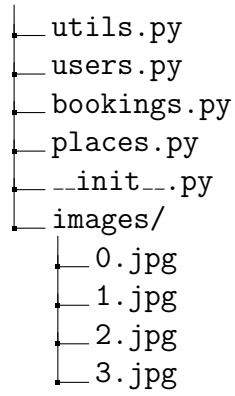
3.5 Arquitetura Implementada

Para fins de facilitar a implementação do projeto, decidimos cortar algumas características da arquitetura que foram mencionadas antes:

- O Broker terá apenas uma fila de eventos, ao invés de várias
- Não teremos exatamente um banco de dados, mas algumas listas em memória, que armazenarão as informações de cada entidade.
- Teremos apenas uma API de ServiçoDeDomínio funcionando como um servidor com 'banco de dados' embutido (processado dentro do mesmo ip, ao invés de ter um endereço específico somente para o BD)
- Não foram implementadas as ideias da seção 3.4.

Os Diagramas de Classe e de Componentes da Implementação se encontram em \uml. Com isso em mente, o funcionamento, entrando em detalhes das tecnologias utilizadas e organização do código, é a seguinte:

```
app/
├── requirements.txt
├── Dockerfile.client
├── Dockerfile.server
├── Dockerfile.broker
├── set_endpoints.js
└── src/
    ├── client.py
    ├── server.py
    ├── broker.py
    └── gui/
    │   ├── config.js
    │   ├── index.html
    │   ├── login.html
    │   ├── menu.html
    │   ├── local.html
    │   ├── topBar.html
    │   ├── topBar.js
    │   ├── administracao.html
    │   └── admin_handlers.js
    └── db/
```



3.5.1 gui/

Referente à interface vista pelo usuário, contempla todo o visual do sistema, com as seções de javascript sendo responsáveis pela realização das requisições, direcionadas ao broker para serem redirecionadas ao servidor.

3.5.2 cliente.py

Simplesmente para servir os arquivos do diretório "gui/" via fastAPI. Por padrão, em localhost:8000 está carregado o arquivo index.html, que redireciona à página inicial do login.html. Para o login, o usuário e a senha são enviados para processamento via POST, que pode retornar 3 resultados:

1. sucesso e usuário é um administrador;
2. sucesso e usuário não é um administrador;
3. falha e o login não ocorre.

Nos casos de sucesso, o usuário é redirecionado à tela de catálogo (menu.html), onde pode ver todos os locais registrados, pesquisar por locais com aplicação de filtros, sair (voltar pra tela de login) e acessar o painel administrativo, caso o usuário seja um administrador.

Um dos arquivos servidos é local.html, que monta a página de um produto dinamicamente com base em seu id, permitindo a criação de reservas para o local selecionado, caso as informações preenchidas estejam consistentes.

Todas as informações sobre usuários, locais e reservas podem ser consultadas/modificadas por qualquer usuário com privilégios administrativos no painel de administração disponibilizado somente para usuários com esse cargo.

3.5.3 broker.py

Implementa uma fila assíncrona, podendo adicionar múltiplos eventos à fila rapidamente. O processamento dos eventos é feito de forma sequencial a fim de evitar double-booking, retirando os eventos da fila e os redirecionando ao respectivo endereço do servidor.

3.5.4 server.py

Onde os eventos emitidos são de fato processados, aceita os verbos GET, POST, PATCH e DELETE como conversão direta para chamar os métodos CRUD das entidades (as funções específicas estão nos respectivos arquivos das entidades, e.g., criar_user() fica em users.py), carrega e transfere imagens (GET /images/{filename}) e faz a validação das informações para login de usuários (POST /login).

3.5.5 db/

Apesar de ser um diretório para agrupar as informações e atuar como banco de dados, via __init__.py, é importado como módulo comum, possuindo as funções de criar, ler, atualizar e deletar usuários, locais e reservas. Possui, também, um diretório específico para o armazenamento das imagens de cada local (limitado a 1 imagem por local na implementação entregue), com mapeamento direto via id: a imagem do local com id 5 possui o nome 5.jpg.

3.5.6 Observações

- Em users.py é manualmente adicionado um usuário administrador padrão na inicialização do código.
- Em places.py foram manualmente adicionados locais fictícios de exemplo, a fim de facilitar a exploração e teste do protótipo desenvolvido.
- Dockerfile.client se utiliza de set_endpoints.sh para criar dinamicamente o arquivo config.js com os endpoints definidos no docker_compose.yaml, para deixar o código de fácil manutenção com endpoints facilmente modificáveis.
- A implementação, por ser um protótipo, não leva em conta questões de segurança, não fazendo tratamento de entrada/sanitização de inputs

menos importantes.