Modelos arquiteturais e suas justificativas

Inicialmente o grupo analisou as alternativas dentre os modelos arquiteturais vistos em sala de aula e chegamos a três os quais consideramos os melhores:

- Microserviços (Microservices)

Motivo:

- Permite que cada tipo de usuário (cliente, restaurante, entregador) tenha seu próprio serviço dedicado.
- Facilita a escalabilidade e manutenção, já que cada funcionalidade (ex: gerenciamento de pedidos, autenticação, pagamentos, etc.) pode ser desenvolvida e implantada separadamente.
- É ideal para sistemas com grande complexidade e com múltiplos fluxos simultâneos.

- MVC (Model-View-Controller)

Motivo:

- Útil para organizar o código da aplicação front-end e back-end, separando interface, lógica e dados.
- Ajuda a manter a clareza e a separação de responsabilidades, especialmente no desenvolvimento do app cliente.

- Camadas (Layered Architecture)

Motivo:

- Ajuda a organizar a aplicação em diferentes camadas: interface (app), lógica de negócio, e persistência (banco de dados).
- Isso torna o sistema mais modular e facilita a manutenção.

- Exemplo de combinação:

- **Front-end** (app dos clientes, restaurantes e entregadores): usa **MVC**.
- **Back-end**: composto por **microserviços** (um para pedidos, um para entregas, um para cadastro de restaurantes, etc.).
- **Cada microserviço** organizado com **camadas** (controle, serviço, repositório).

Padrão Arquitetural Adotado: MVC (Model-View-Controller)

Justificativa:

1. Modularidade

- O padrão MVC separa o sistema em três camadas bem definidas:
  - **Model**: regras de negócio e dados.
  - **View**: interface com o usuário.
  - **Controller**: lógica de controle e fluxo entre Model e View.
- Isso facilita o desenvolvimento paralelo por diferentes membros da equipe e permite alterar uma parte sem afetar as outras.

2. Manutenibilidade

- Com o código separado por responsabilidades, é mais fácil localizar, entender e corrigir erros.

3. Reusabilidade

- Os **Models** (como Pedido, Restaurante, Produto) podem ser reutilizados entre diferentes interfaces (ex: app do cliente e painel do dono do restaurante).

4. Escalabilidade

- Embora o MVC por si só não seja distribuído, ele organiza o sistema de forma que partes específicas possam ser isoladas e escaladas posteriormente.
- Por exemplo: se o módulo de pedidos crescer, pode ser facilmente extraído para um serviço separado no futuro.

5. Facilidade de Testes

- A separação entre lógica de negócio, controle e interface acilita testes unitários e de integração.
- Pode-se testar a lógica dos pedidos ou cálculo de taxas sem depender da interface gráfica.

6. Usabilidade

- Como a View está separada da lógica, a equipe pode focar em criar uma experiência de usuário fluida sem interferir no backend.
- Permite criação de interfaces diferentes para cada perfil de usuário (Cliente, Dono, Entregador), com base nas mesmas regras de negócio.

7. Flexibilidade

- Mudanças na interface, como design ou layout, não afetam a lógica de negócios nem os dados.
- É possível adaptar o mesmo backend para diferentes interfaces (ex: app Android, iOS e painel web).

Resumo

A adoção do padrão **MVC** traz uma organização clara que melhora a **modularidade**, **manutenibilidade** e **testabilidade** do sistema, além de permitir **reuso de componentes** e facilitar o desenvolvimento de interfaces para diferentes tipos de usuários. Essa estrutura é especialmente valiosa para um sistema com múltiplos perfis e funcionalidades como o app do Ifood proposto.