| **Universidade Federal Fluminense**  Projeto de Software |
| --- |
| **Minha Lanchonete** |

Documento de Arquitetura

**Diretriz Arquitetural**

| **Alunos do projeto** |
| --- |
| *Arthur Pinheiro Machado* |
| *Willian Rodrigues Benevides e Silva* |
|  |

**Sumário**

**INTRODUÇÃO ...................................................................................................................... 4  
1.1 Escopo do Sistema ........................................................................................................4  
1.2 Funcionalidades Principais ...........................................................................................5  
2.1 Descrição dos Requisitos Arquiteturais .......................................................................6  
2.2 Objetivos da Arquitetura .............................................................................................. 7  
2.3 Restrições da Arquitetura ............................................................................................. 8  
3.1 Definição dos Padrões Arquiteturais Adotados ..........................................................9  
3.2 Justificativa com base nos atributos de qualidade....................................................12  
3.1.1 Manutenibilidade .................................................................................................... 12  
3.1.2 Escalabilidade ........................................................................................................ 13  
3.1.3 Segurança ............................................................................................................... 13  
3.1.4 Confiabilidade ........................................................................................................ 14  
3.1.5 Usabilidade (API) .................................................................................................. 14  
3.1.6 Reutilização ........................................................................................................... 15**

**4.1 Justificativa de uso dos estilos arquiteturais utilizados…………………………….15**

**5.1 Aplicação dos princípios SOLID e padrões GRASP………………………………… 15**

**6.1 Adoção de padrão GoF…………………………………………………………………….15**

**7.1 Evolução para Linha de Produto de Software………………………………………….16**

**REFERÊNCIAS .................................................................................................................. 16**

**1. Escopo do Sistema**

O sistema tem como objetivo gerenciar os pedidos realizados em uma lanchonete, permitindo o registro de produtos, gerenciamento de clientes, realização de pedidos, gerenciamento de métodos de pagamento e acompanhamento do histórico de vendas.

Este sistema possibilitará que atendentes ou operadores realizem pedidos associando produtos ao cliente e ao método de pagamento escolhido, registrando data e hora do pedido para posterior acompanhamento, relatórios e controle de vendas.

O sistema também deve incluir o papel do entregador, que retira pedidos de uma rota de entrega e concluí os pedidos conforme for entregando.

**1.1 Funcionalidades principais:**

**Cadastro e Gerenciamento de Produtos**

* Produtos genéricos (pizzas, bebidas e outros) com imagem, nome e valor.
* Informações específicas de produtos, como volume (bebidas) e tamanho/descrição (pizzas).

**Cadastro e Gerenciamento de Clientes**

* Cadastro de dados de clientes necessários para pedidos.

**Cadastro e Gerenciamento de Métodos de Pagamento**

* Permite associar métodos de pagamento a clientes (cartões, pix, etc.).

**Realização e Gerenciamento de Pedidos**

* Criação de pedidos com data e hora de realização.
* Associação de pedidos a clientes e métodos de pagamento.
* Adição de múltiplos produtos em um pedido, com possibilidade de especificar quantidades.

**Consulta e Relatórios de Pedidos**

* Listagem de pedidos realizados, filtragem por data ou cliente.
* Visualização dos produtos incluídos em cada pedido.

**Escalabilidade para funcionalidades futuras:**

* Integração de funcionalidades de estoque.
* Gestão de entrega de pedidos e controle de status (em preparo, pronto, entregue).
* Geração de relatórios financeiros e de vendas mensais.

**2.1 Descrição dos Requisitos Arquiteturais**

Os requisitos arquiteturais do sistema garantem que o nosso projetoatenda de forma adequada aos aspectos de desempenho, manutenibilidade e escalabilidade esperados para o projeto. São eles:

**Separação de Camadas:**

* Separação entre camada de apresentação, camada de aplicação, camada de domínio (modelos) e camada de persistência de dados.
* Facilitar manutenções, testes e mudanças na interface sem afetar a lógica de negócios.

**Persistência de Dados Relacional:**

* Uso de banco de dados relacional (PostgreSQL) para armazenamento de clientes, produtos, pedidos e métodos de pagamento, garantindo integridade referencial.

**Acesso Confiável aos Dados:**

* Controle de integridade no nível de banco de dados e regras de negócios na aplicação, evitando inconsistências em pedidos e cadastros.

**Modularidade**

* Organização em aplicações modulares dentro do projeto Django:
  + products (produtos, pizzas e bebidas)
  + orders (pedidos e itens do pedido)
  + users (clientes)
  + payments (métodos de pagamento)

**Escalabilidade para API REST**

* Estrutura de endpoints RESTful para integração com sistemas de autoatendimento ou painéis administrativos no futuro.

**2.2 Objetivos da Arquitetura**

* **Facilitar a manutenção e evolução do sistema**, permitindo a adição de funcionalidades futuras (controle de estoque, relatórios avançados, controle de entregas).
* **Garantir desempenho adequado**, suportando múltiplos pedidos sendo realizados de forma concorrente.
* **Garantir segurança e integridade dos dados**, evitando pedidos com dados inconsistentes ou pagamentos sem método válido.
* **Permitir rastreabilidade de pedidos e vendas**, garantindo auditoria de dados (quando o pedido foi feito, por quem, o que foi comprado).
* **Facilitar testes automatizados**, utilizando a estrutura de testes do Django.
* **Facilitar implantação local ou em nuvem** (Heroku, Railway, etc.) para demonstrações ou uso real.

**2.2 Restrições da Arquitetura**

* **Uso de Banco Relacional**, não sendo possível utilizar bancos NoSQL neste primeiro momento, para manter a integridade e facilitar o uso de ForeignKeys.
* **Escopo limitado ao controle de pedidos**, sem implementação de funcionalidades de delivery em tempo real ou integrações com gateways de pagamento no momento inicial.
* **Interface Web Simples** (admin ou páginas básicas) sem foco inicial em aplicativo mobile nativo.
* **Infraestrutura com recursos limitados** caso usado localmente ou em serviços gratuitos na nuvem, limitando a escalabilidade inicial.

**3.1 Definição dos Padrões Arquiteturais Adotados**

### Arquitetura em Camadas (Layered Architecture)

O sistema segue Arquitetura em Camadas, separando responsabilidades para organização e manutenibilidade:

* **Camada de Apresentação (Interface):**Responsável por expor a interface REST para o usuário via *views* do Django REST Framework.
* **Camada de Aplicação:**Implementada nas *views*, gerenciando fluxos de negócio como:
  + Controle de permissões (UserHelper).
  + Lógicas de criação de pedidos associando produtos e métodos de pagamento.
  + Regras específicas, como somente administradores podem criar produtos, apenas clientes podem criar pedidos.
* **Camada de Domínio:**Representada pelos *models*: Product, Drink, Pizza, Order, PaymentMethod, etc., encapsulando os dados e garantindo a integridade do domínio.
* **Camada de Persistência:**Gerenciada pelo ORM do Django, permitindo consultas, criação e atualizações em um banco relacional de forma abstrata e segura.

### MVC (Model-View-Controller)

O Django REST Framework adapta o **padrão MVC**, onde:

* **Model:** Responsável por armazenar e gerenciar os dados do sistema (models da lanchonete).
* **View (Controller):** Gerencia requisições HTTP, permissões e chamadas aos serializers para fornecer respostas JSON.
* **Template (que não foi utilizado):** Seria a camada de apresentação HTML, mas aqui substituída por respostas REST.

### Padrão RESTful

A API segue o **padrão RESTful**, estruturando endpoints em torno de recursos:

* ProductView, PizzaView, DrinkView, OrderView, PaymentMethodView
* Uso de verbos HTTP para definir operações:
  + GET ➔ Listar ou obter recurso.
  + POST ➔ Criar recurso.
  + PUT/PATCH ➔ Atualizar recurso.
  + DELETE ➔ Remover recurso.

Isso facilita a integração com front-ends diversos (mobile/web), clareza e consistência nos endpoints e a testabilidade com ferramentas como Postman.

### Padrão Repository (implícito via ORM Django)

As *views* utilizam o ORM do Django para acessar dados desacoplados do banco de dados, realizando:

* .objects.create()
* .objects.filter()
* .get\_object()

Seguindo implicitamente o padrão **Repository**, mantendo o acesso aos dados encapsulado.

### Padrão Serializer

Os Serializers do Django REST Framework seguem o padrão DTO (Data Transfer Object), transformando os dados dos modelos em JSON de forma estruturada e segura:

* Encapsulam a lógica de formatação e transformação de dados.
* Possibilitam campos calculados por SerializerMethodField para retornar dados agregados (ex: nome e valor de Product em DrinkSerializer e PizzaSerializer).
* Auxiliam na separação entre representação de dados e lógica de negócio.

### Padrão de Autorização Baseada em Função

Por meio de UserHelper:

* Restringe ações de criação e atualização a administradores, como criar pizzas, drinks, editar produtos.
* Restringe ações específicas a clientes, como criar pedidos, associar métodos de pagamento.
* Retorna status HTTP 403 quando permissões não são atendidas, mantendo integridade e segurança.

### Padrão DRY (Don't Repeat Yourself)

* GenericViewSet com *mixins* (CreateModelMixin, ListModelMixin, etc.)
* Serializers reutilizados em diferentes *views*
* Permissões centralizadas via UserHelper reduz duplicações e facilita a manutenção do sistema.

**3.2 Justificativa das decisões arquiteturais com base nos atributos de qualidade**

As decisões arquiteturais tomadas no projeto foram fundamentadas visando garantir que o sistema seja confiável, manutenível, extensível e seguro para o contexto de uso em uma lanchonete.

#### 3.2.1 Manutenibilidade

**Decisões Arquiteturais Relacionadas:**

* Uso de **Arquitetura em Camadas** separando:
  + Interface REST (Views)
  + Lógica de aplicação (fluxos de negócio nas views)
  + Domínio (models)
  + Persistência (ORM)
* Uso de **Django REST Framework com mixins**, evitando duplicação de código.
* Organização modular do sistema (products, orders, payments, users).

Facilita:

* Correções de erros sem impactar outros módulos.
* Evolução incremental (ex: adicionar controle de estoque, relatórios de vendas).
* Localização rápida de funcionalidades específicas.

#### 3.2.1 Escalabilidade

**Decisões Arquiteturais Relacionadas:**

* Utilização de **RESTful API** para permitir fácil integração com novos front-ends (ex: aplicativo mobile, quiosques de autoatendimento).
* Uso do **ORM do Django**, que possibilita trocar o banco sem alterar a lógica do sistema.
* Separação entre os domínios (produtos, pedidos, pagamentos), permitindo evolução independente.

Permite:

* Suporte a mais usuários e pedidos simultâneos conforme o crescimento da lanchonete.
* Integração futura com sistemas de entrega ou gateways de pagamento sem refatorações pesadas.

#### 3.1.3 Segurança

**Decisões Arquiteturais Relacionadas:**

* Uso de IsAuthenticated em todas as views, garantindo que apenas usuários autenticados utilizem a API.
* Controle de permissões via UserHelper:
  + Restringe algumas ações apenas para administradores.
  + Restringe acesso a dados sensíveis de acordo com o tipo de usuário

Garante:

* Somente usuários autorizados conseguem realizar ações específicas.
* Dados dos pedidos e pagamentos são acessados apenas pelo cliente ou administrador.
* Previne modificações não autorizadas.

#### 3.1.4 Confiabilidade

**Decisões Arquiteturais Relacionadas:**

* Uso do **ORM do Django** para garantir integridade transacional ao realizar pedidos e associar produtos.
* Validação de permissões e consistência dos dados nas *views* por exemplo: método de pagamento pertence ao cliente?.
* Estrutura REST clara, retornando códigos de status apropriados (403, 400, 500).

Contribuição:

* Evita inserção de dados inconsistentes (ex: pagamento sem cliente válido).
* Permite rastrear erros facilmente.
* Garante execução correta dos fluxos de pedido e cadastro de produtos.

#### 3.1.5 Usabilidade (API)

**Decisões Arquiteturais Relacionadas:**

* Uso de **serializers claros** para controlar a estrutura e a legibilidade dos dados enviados e recebidos.
* Respostas REST padronizadas, facilitando o consumo por front-ends diversos.

**Contribuição:**

* É possível integrar front-ends facilmente usando Postman ou clients HTTP.
* Permite fácil uso em um sistema de autoatendimento no futuro.

#### 3.1.6 Reutilização

**Decisões Arquiteturais Relacionadas:**

* Utilização de *mixins* e GenericViewSet no Django REST Framework.
* Reaproveitamento de serializers em diferentes *views*.
* Modularização em apps (products, orders, payments).

**Contribuição:**

* Componentes e lógicas podem ser utilizados em outros sistemas similares.
* Facilidade para evoluir funcionalidades (ex: cadastro de sanduíches reutilizando Product)

**4.1 Justificativa de uso dos estilos arquiteturais utilizados**

* **Arquitetura em Camadas**Separa o sistema em apresentação, aplicação, domínio e persistência, facilitando a manutenção, testes e evolução independente de cada parte. Essa organização melhora a manutenibilidade e a clareza do projeto.
* **MVC Adaptado pelo Django REST Framework**As views atuam como controladores das requisições HTTP, os models gerenciam os dados, e os serializers substituem templates HTML para gerar respostas JSON. Essa abordagem organiza bem as responsabilidades e favorece a reutilização.
* **API RESTful**A API é organizada em torno de recursos e utiliza verbos HTTP para as operações, o que facilita a integração com diversos front-ends, testes automatizados e manutenção.
* **Padrão Facade (UserHelper)**Centraliza a lógica de verificação de permissões (admin, cliente) em uma única classe, reduzindo duplicação e acoplamento entre views e regras de negócio, simplificando a manutenção.
* **Padrão Repository (ORM do Django)**O ORM abstrai o acesso e manipulação do banco de dados relacional (PostgreSQL), possibilitando mudanças no banco sem impactar a lógica da aplicação e garantindo integridade dos dados.
* **Data Transfer Object (DTO) via Serializers**Os serializers controlam a validação, formatação e exposição dos dados em JSON, separando a representação dos dados da lógica interna e garantindo segurança e flexibilidade na API.

**5.1 Aplicação dos princípios SOLID e padrões GRASP**

#### 5.2 Single Responsibility Principle (SRP)

**Princípio:** Cada classe deve ter uma única responsabilidade, não misturando lógicas distintas.

**Aplicação:**

* A model User é responsável apenas pelo gerenciamento de dados de autenticação e atributos básicos.
* Models Client e Admin armazenam informações específicas de cada tipo de usuário (CPF, endereço, PIS/PASEP, data de admissão) separadamente.
* Serializers UserSerializer, ClientSerializer e AdminSerializer são especializados em formatar os dados específicos para cada entidade, evitando serializers genéricos e confusos.

**Refatoração:**

* Inicialmente, campos como cpf e pis\_pasep estavam sendo manipulados diretamente em views.
* Foram centralizados como propriedades no model User, que delega ao Client ou Admin automaticamente, evitando repetição de lógica e tornando as views mais simples.

#### 5.3 Open/Closed Principle (OCP)

**Princípio:** O sistema deve ser aberto para extensão e fechado para modificação.

**Aplicação:**

* Para incluir novos tipos de usuário (ex: Funcionário), basta criar um novo model e serializer, sem precisar modificar o User.
* Views (AdminsView, UsersView, RegisterView) foram implementadas de forma genérica usando GenericViewSet e mixins, permitindo facilmente adicionar novos métodos ou endpoints sem alterar os existentes.

#### 5.4 Liskov Substitution Principle (LSP)

**Princípio:** Subtipos podem substituir seus tipos base sem alterar a funcionalidade do sistema.

**Aplicação:**

* O User pode ser manipulado genericamente nas views, sendo substituído por usuários do tipo Admin ou Client nas operações, sem quebrar o funcionamento das views.
* Serializers UserAdminSerializer e UserClientSerializer podem substituir UserSerializer onde necessário, garantindo compatibilidade.

#### 5.5 Interface Segregation Principle (ISP)

**Princípio:** Interfaces específicas são melhores do que interfaces gerais.

**Aplicação:**

* Os serializers foram separados por tipo (AdminSerializer, ClientSerializer, UserAdminSerializer, UserClientSerializer), evitando campos desnecessários em retornos de endpoints.
* Cada view (AdminsView, UsersView, LoginView, RegisterView) utiliza apenas as permissões e métodos necessários, evitando implementar métodos HTTP desnecessários.

#### 5.6 Dependency Inversion Principle (DIP)

**Princípio:** Dependa de abstrações, não de implementações.

**Aplicação:**

* O sistema utiliza abstrações do Django REST Framework (GenericViewSet, mixins, serializers) para isolar lógica de requisições e persistência.
* O UserHelper abstrai a lógica de verificação de papéis (is\_admin, is\_client), permitindo que as views dependam de abstrações em vez de checagens manuais repetitivas.

### 5.7 Padrões GRASP

#### 

#### 5.8 Creator

**Aplicação:**

A responsabilidade de criação de Admin e Client foi atribuída às respectivas views e serializers, que possuem as informações necessárias para instanciar corretamente os objetos.

#### 5.9 Controller

**Aplicação:**

* Views como AdminsView, UsersView e RegisterView atuam como controladores, recebendo as requisições HTTP, realizando as verificações de permissão e delegando ao serializer ou ORM as operações necessárias.

#### 5.10 Information Expert

**Aplicação:**

* Serializers (AdminSerializer, ClientSerializer, etc.) são responsáveis por formatar e expor os dados necessários, sendo especialistas no formato dos dados retornados pela API.
* Models User, Admin e Client encapsulam os dados específicos de cada entidade, mantendo a responsabilidade pelas informações que possuem.

#### 5.11 Low Coupling

**Aplicação:**

* Separação clara entre camada de apresentação (views), lógica de domínio (models) e formatação de dados (serializers).
* UserHelper desacopla a verificação de permissões das views, permitindo mudanças nesta lógica sem afetar as views.

**5.12 High Cohesion**

**Aplicação:**

* Cada classe possui um único propósito:
  + **Models**: representação do domínio.
  + **Serializers**: formatação e validação de dados.
  + **Views**: controle de requisições e roteamento de fluxos.
* Organização modular (views, serializers, models, helpers) reforça a coesão do sistema.

**6.1 Adoção de Padrão GoF no projeto**

### Padrão GoF adotado: Facade

O padrão **Facade** é utilizado para simplificar o acesso a subsistemas complexos, fornecendo uma interface unificada para um conjunto de interfaces de um subsistema, facilitando o uso e a redução do acoplamento entre as partes do sistema.

### Aplicação prática no projeto:

O módulo UserHelper atua como uma Facade para o sistema de verificação de permissões de usuários, encapsulando a lógica para verificar se um usuário é administrador ou cliente.

#### Sem o padrão Facade:

As views precisariam implementar repetidamente a lógica de verificação, acessando diretamente relacionamentos do usuário:

if user.admin is not None:

# é admin

ou

try:

user.admin

# é admin

except:

# não é admin

Isto geraria alta duplicação e acoplamento entre views e detalhes de implementação do domínio.

#### Com o padrão Facade:

Foi criada a classe **UserHelper**, que encapsula toda a lógica de verificação de papéis:

class UserHelper:

@staticmethod

def is\_admin(user):

return hasattr(user, 'admin')

@staticmethod

def is\_client(user):

return hasattr(user, 'client')

As views utilizam essa interface simplese limpa:

if not UserHelper.is\_admin(request.user):

return Response(status=403)



**7.1 Evolução para linha de produto de software (LPS)  
  
Modelo de Características (Feature Model)**

### 7.2. Características Mandatórias (obrigatórias):

São comuns a todas as variantes do sistema.

* **Gerenciamento de Usuários**
  + Cadastro de clientes
  + Cadastro de administradores
  + Login / Autenticação JWT
* **Gerenciamento de Produtos**
  + Cadastro e edição de produtos
* **Gerenciamento de Pedidos**
  + Criação e acompanhamento de pedidos
* **Gerenciamento de Métodos de Pagamento**
  + Cadastro e vinculação de métodos de pagamento aos clientes
* **API RESTful**
* **Camada de Autorização**
* **Auditoria de Ações**
* **Persistência de dados com Banco Relacional**

### 7.3. Características Opcionais:

Podem ser incluídas conforme a necessidade do cliente ou versão do sistema.

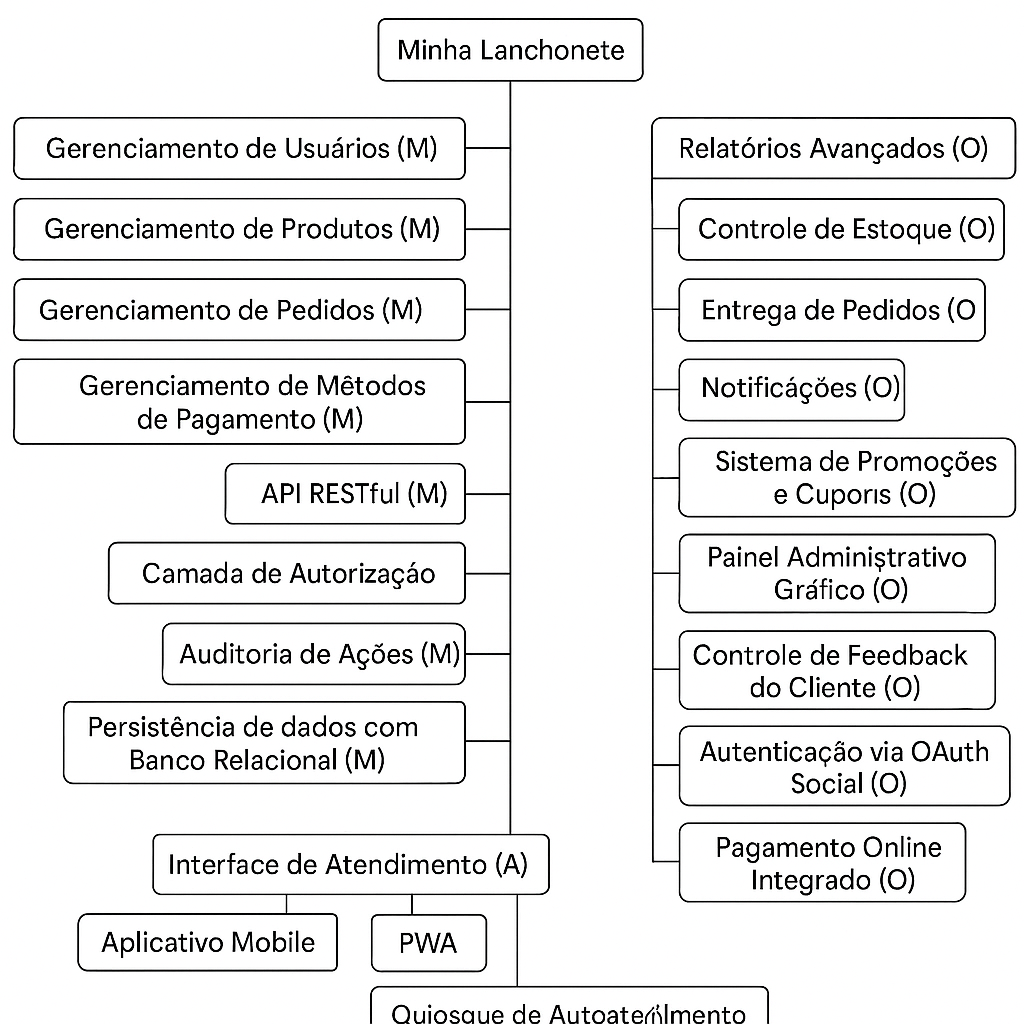
* **Relatórios Avançados**
  + Vendas por período
  + Produtos mais vendidos
* **Controle de Estoque**
* **Entrega de Pedidos (Delivery)**
* **Notificações (e-mail ou push)**
* **Sistema de Promoções e Cupons**
* **Painel Administrativo Gráfico (Dashboard)**
* **Controle de Feedback do Cliente**
* **Autenticação via OAuth Social (Google/Facebook)**
* **Pagamento Online Integrado (ex: PIX, cartão via API)**

### 7.4 Características Alternativas:

São mutuamente exclusivas dentro de um grupo.

* **Interface de Atendimento:**
  + Aplicativo Mobile (Flutter)
  + PWA (Progressive Web App)
  + Quiosque de Autoatendimento (Totem)
* **Tipos de Produtos:**
  + Pizzas
  + Bebidas
  + Sanduíches

Em um cliente pode-se ativar apenas pizzas e bebidas, enquanto outro pode incluir sanduíches dependendo do tipo de estabelecimento.



Legenda:

* **(M)**: Mandatório
* **(O)**: Opcional
* **(A)**: Alternativo

Permite configurar diferentes produtos da LPS, ajustando:

* Recursos disponíveis por cliente.
* Interface de atendimento.
* Tipo de produtos suportados.

Facilita reuso de código, padronização e manutenção, reduzindo custos de desenvolvimento ao criar variantes para diferentes estabelecimentos.

Permite estratégias de monetização diferenciadas, oferecendo planos básicos (somente funcionalidades mandatórias) e avançados (com funcionalidades opcionais).

Facilita a gestão de variabilidades no futuro, utilizando frameworks como Django Settings**,** Feature Flags e Modularização por Apps no repositório.