GearPizza Flutter App – Technical Architecture Documentation

Table of Contents

Project Overview

Application Purpose:

GearPizza is a comprehensive pizza ordering platform. This project is basically hiring assignment for Revo Digital. This platform has dual interfaces:

- **Customer Interface**: Browse restaurants, view pizzas, manage cart, place orders
- **Owner Interface**: Manage Pizzas, track orders, update order status

Technical Stack:

- **Framework:** Flutter 3.x
- **State Management:** Provider pattern
- **Backend Integration**: Directus CMS. Right now only mock fallbacks (not completely integrated)
- **External APIs:** Google Places API for address autocomplete
- **Local Storage:** In-memory state management
- **UI Framework:** Material Design 3

Pre-Development Decisions

Before starting development, I made several key decisions about how to structure and build the GearPizza app. These choices were based on my Flutter knowledge, project requirements, and practicality.

Architecture Philosophy

Decision: Clean Architecture with layered separation

- **Easy to understand:** Clear separation between UI, business logic, and data makes the code easier to follow

- **Better organization:** Each part of the app has its own responsibility. Screens handle UI, providers handle state, services handle data
- **Testing friendly:** I can test each part separately without worrying about the others
- **Switching data sources:** Easy to switch between real API data and mock data for testing

I decided to focus on the core features first and put my maximum effort on them also because Flutter is a framework I had basic familiarity with. I familiarized myself with Directus, but I put the advanced feature of integrating Directus at the end so I can develop the core features as flawless as possible. In a private repository I did try to integrate Directus, but there are some bugs I need to care of hence I rolled back to core feature version and so far, sent Revo team that repository.

How it works in practice:

- **Screens** → Handle what users see and interact with
- **Providers** → Manage app state and business logic
- **Services** → Get data from APIs or create mock data
- **Models** → Define what our data looks like

State Management Choice

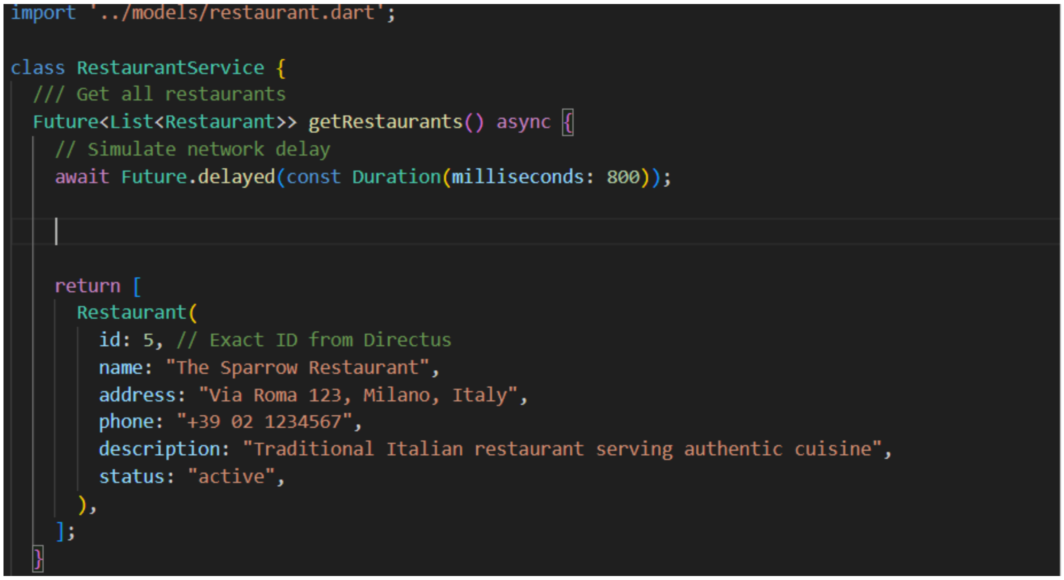
Decision: Provider pattern. Not too complex as Bloc/Riverpod and not too simple as setState

- **Familiar concepts:** I already understood ChangeNotifier from Flutter basics, so Provider was a natural next step
- **Simple to implement:** Less boilerplate code compared to Bloc. I could focus on building features instead of complex setup
- **Good Flutter integration:** Works naturally with Flutter's widget system and hot reload
- **Perfect for this app size:** The app isn't complex enough to need heavy-duty state management

Data Strategy

Decision: Mock-first development with API integration points

- **No backend blocking:** I could build the entire frontend without depending on backend.
- **Reliable demos:** Mock data means the app always works during presentations.
- **Easy testing:** I can control exactly what data to test with. Mock data was entirely identical to data on Directus pod provided by Revo.



```
import '../models/restaurant.dart';

class RestaurantService {
  /// Get all restaurants
  Future<List<Restaurant>> getRestaurants() async {
    // Simulate network delay
    await Future.delayed(const Duration(milliseconds: 800));



    return [
      Restaurant(
        id: 5, // Exact ID from Directus
        name: "The Sparrow Restaurant",
        address: "Via Roma 123, Milano, Italy",
        phone: "+39 02 1234567",
        description: "Traditional Italian restaurant serving authentic cuisine",
        status: "active",
      ),
    ];
  }
```

Navigation Strategy

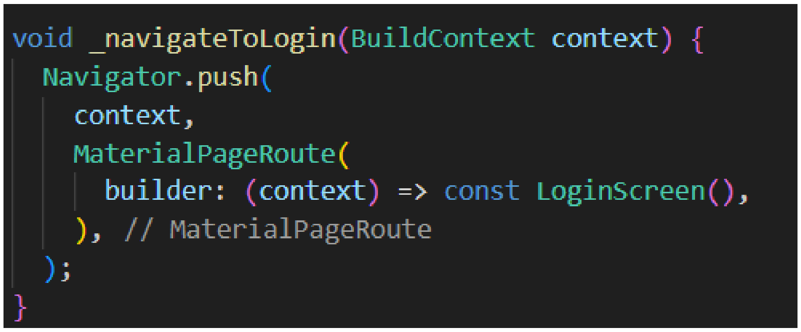
Decision: Imperative navigation with MaterialPageRoute

- **Direct control:** I can see exactly what's happening when navigating between screens
- **Easy debugging:** When something goes wrong, it's clear which navigation call caused it
- **Less complexity:** No need to hassle with complex routing packages

- **Flutter standard:** Using Flutter's built-in navigation patterns

Navigation patterns I used:

- **Push:** For normal screen-to-screen navigation
- **Push replacement:** For login flows where users shouldn't go back
- **Pop:** For going back or closing modals
- **Modal bottom sheets:** For pizza details and forms



```
void _navigateToLogin(BuildContext context) {
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => const LoginScreen(),
    ), // MaterialPageRoute
  );
}
```

UI/UX Decisions

Decision: Material Design 3 with custom colors

- **Consistent look:** Users are familiar with Material Design patterns
- **Built-in accessibility:** Material components have accessibility features built-in
- **Less design work:** I could focus on functionality instead of designing everything from scratch
- **Easy theming:** Simple color changes gave the app a pizza/food brand feel

Custom touches:

- **Pizza red:** Color.fromARGB(255, 200, 45, 45) for primary brand color
- **Warm background:** Cream color for a food-friendly feel
- **Consistent spacing:** Used Flutter's standard spacing throughout

These decisions helped me build a functional, well-organized app while keeping things manageable.

The choices prioritized:

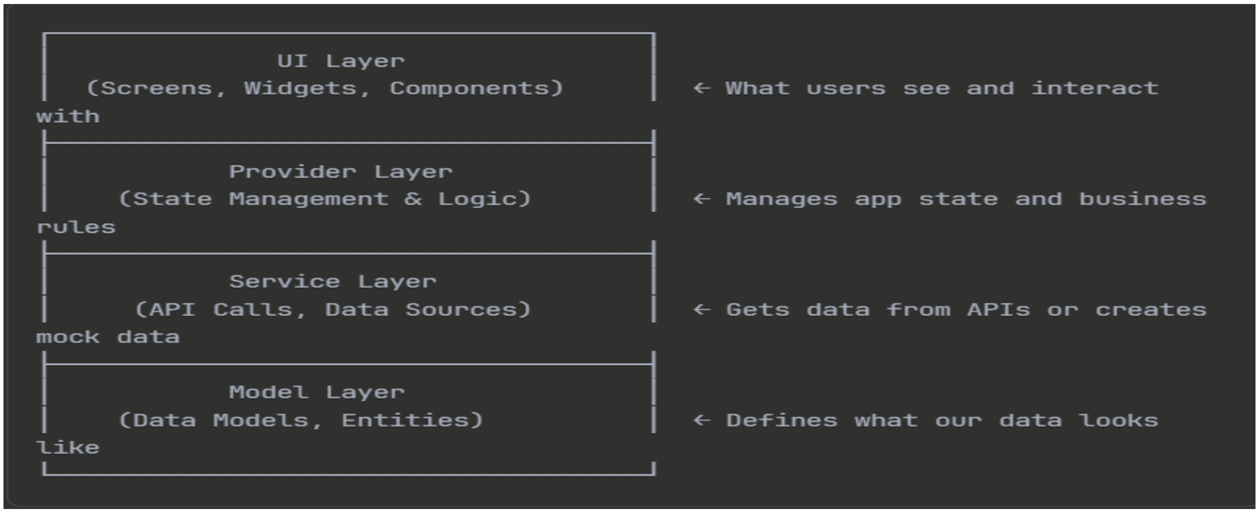
- **Learning:** Using patterns I could understand and explain
- **Practicality:** Solutions that work reliably without over-engineering
- **Maintainability:** Code structure that's easy to modify and extend
- **Demo-ability:** An app that always works for presentations

---

Architecture Overview

How I Organized the App Structure

I organized my app into clear layers so that each part has a specific job. This makes the code easier to understand, debug, and modify. Here's how it works:



```
┌──────────────────────────────────────┐
│              UI Layer                │
│   (Screens, Widgets, Components)     │  ← What users see and interact
with
├──────────────────────────────────────┤
│            Provider Layer            │
│     (State Management & Logic)       │  ← Manages app state and business
rules
├──────────────────────────────────────┤
│            Service Layer             │
│      (API Calls, Data Sources)       │  ← Gets data from APIs or creates
mock data
├──────────────────────────────────────┤
│             Model Layer              │
│      (Data Models, Entities)         │  ← Defines what our data looks
like
└──────────────────────────────────────┘
```

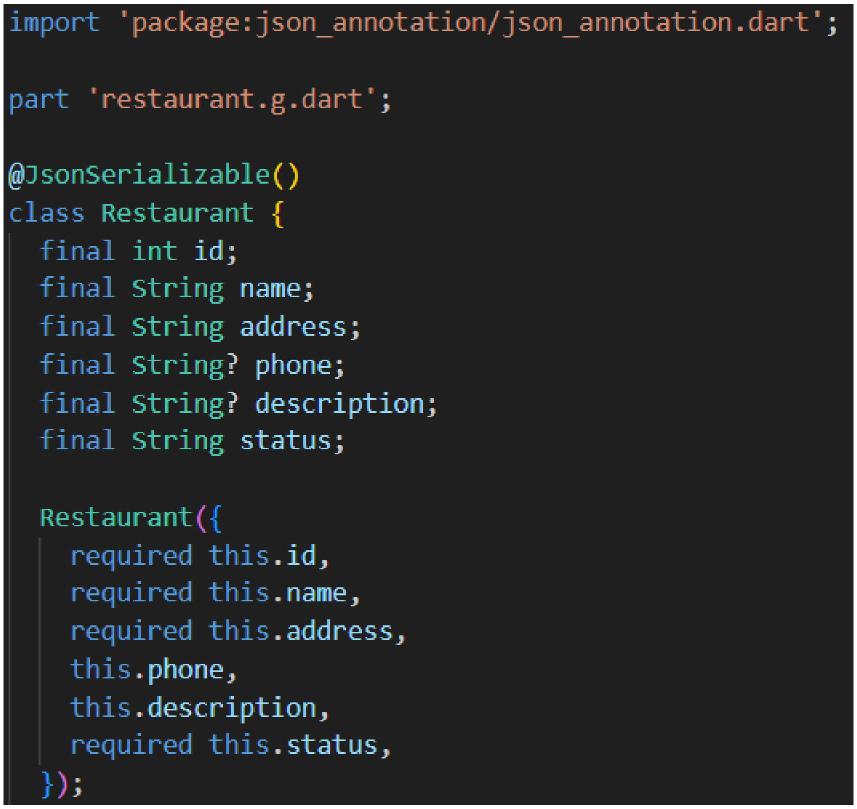
- **Easy to find things:** If I need to fix a UI bug, I look in screens. If there's a data issue, I check services

- **One responsibility per layer:** Each layer has one clear job, making code easier to understand
- **Easy to test:** I can test business logic separately from UI, and UI separately from data fetching

Key Principles I followed

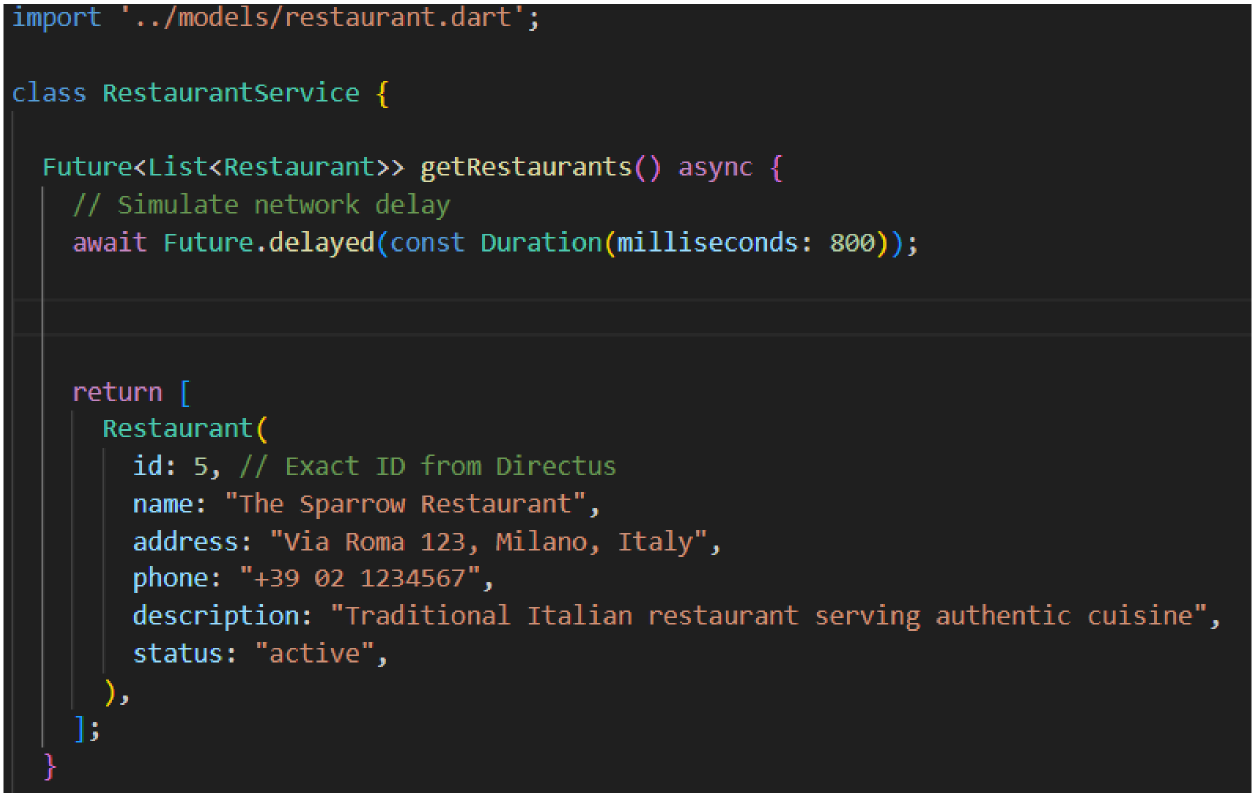
Each part of the app has its own responsibility

Models (Data Structures):



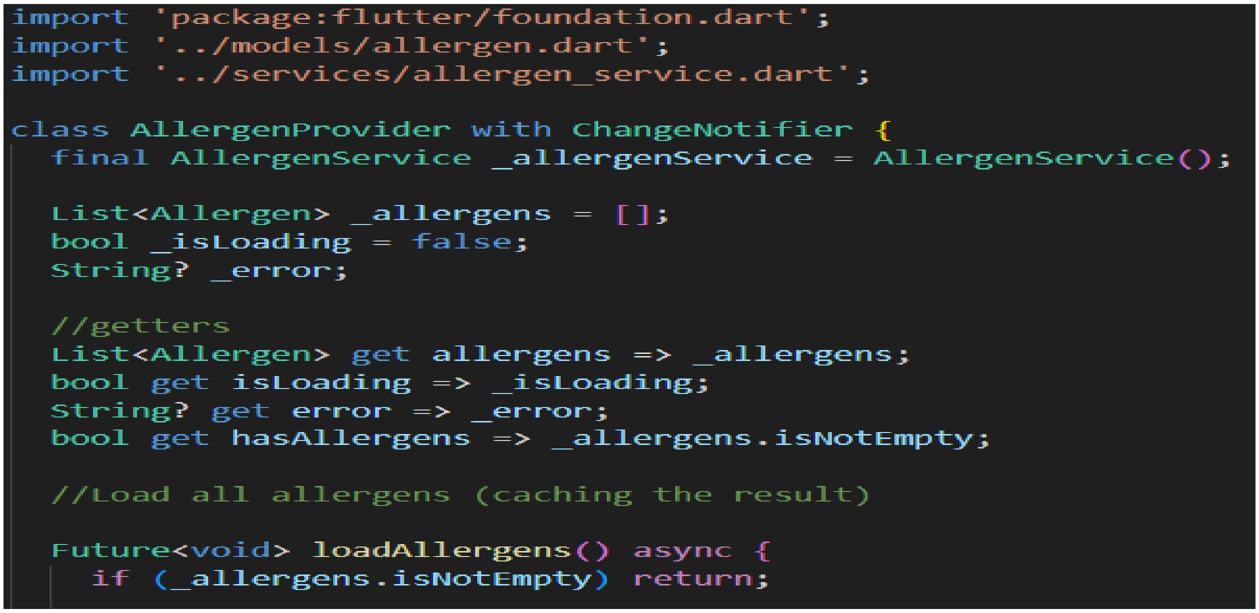
```dart
import 'package:json_annotation/json_annotation.dart';

part 'restaurant.g.dart';

@JsonSerializable()
class Restaurant {
  final int id;
  final String name;
  final String address;
  final String? phone;
  final String? description;
  final String status;

  Restaurant({
    required this.id,
    required this.name,
    required this.address,
    this.phone,
    this.description,
    required this.status,
  });
```

Services (Data Operations):



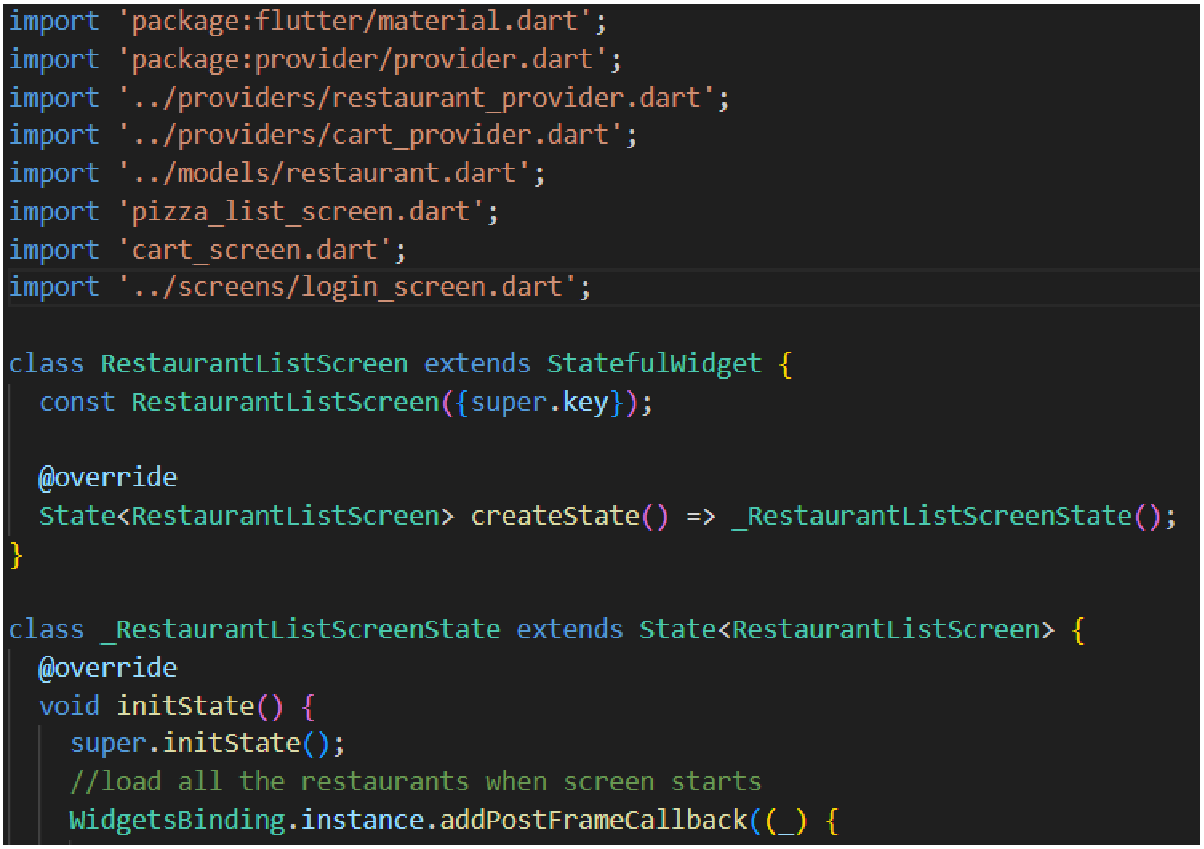
```dart
import '../models/restaurant.dart';

class RestaurantService {

  Future<List<Restaurant>> getRestaurants() async {
    // Simulate network delay
    await Future.delayed(const Duration(milliseconds: 800));



    return [
      Restaurant(
        id: 5, // Exact ID from Directus
        name: "The Sparrow Restaurant",
        address: "Via Roma 123, Milano, Italy",
        phone: "+39 02 1234567",
        description: "Traditional Italian restaurant serving authentic cuisine",
        status: "active",
      ),
    ];
  }
```

Providers (State & Business Logic):



```dart
import 'package:flutter/foundation.dart';
import '../models/allergen.dart';
import '../services/allergen_service.dart';

class AllergenProvider with ChangeNotifier {
  final AllergenService _allergenService = AllergenService();

  List<Allergen> _allergens = [];
  bool _isLoading = false;
  String? _error;

  //getters
  List<Allergen> get allergens => _allergens;
  bool get isLoading => _isLoading;
  String? get error => _error;
  bool get hasAllergens => _allergens.isNotEmpty;

  //Load all allergens (caching the result)

  Future<void> loadAllergens() async {
    if (_allergens.isNotEmpty) return;
```

UI (Presentation Only):



```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '../providers/restaurant_provider.dart';
import '../providers/cart_provider.dart';
import '../models/restaurant.dart';
import 'pizza_list_screen.dart';
import 'cart_screen.dart';
import '../screens/login_screen.dart';

class RestaurantListScreen extends StatefulWidget {
  const RestaurantListScreen({super.key});

  @override
  State<RestaurantListScreen> createState() => _RestaurantListScreenState();
}

class _RestaurantListScreenState extends State<RestaurantListScreen> {
  @override
  void initState() {
    super.initState();
    //load all the restaurants when screen starts
    WidgetsBinding.instance.addPostFrameCallback((_) {
```

Simple Dependency Flow

I kept dependencies flowing in one direction to avoid confusion:

UI → Provider → Service → External APIs/Mock Data

How it works:

1. **UI** asks **Provider** for data
2. **Provider** asks **Service** to fetch data
3. **Service** gets data from API or mock source
4. Data flows back up the chain to update the UI

State Management Implementation

Provider System:

I created four main providers to handle different parts of the app's state. Each provider has a clear purpose and manages its own piece of the puzzle.

RestaurantProvider – Managing Restaurant Data:

Below is a snippet of code with explanation.(The code is just a snippet, so might not be complete in this document, only for reference. The main point is to explain my state management logic)

```
import 'package:flutter/foundation.dart';
import '../models/restaurant.dart';
import '../services/restaurant_service.dart';

class RestaurantProvider with ChangeNotifier {
  //private variables

  final RestaurantService _restaurantService = RestaurantService();
  List<Restaurant> _restaurants = [];
  Restaurant? _selectedRestaurant;
  bool _isLoading = false;
  String? _error;

  //public getter
  List<Restaurant> get restaurants => _restaurants;
  Restaurant? get selectedRestaurant => _selectedRestaurant;
  bool get isLoading => _isLoading;
  String? get error => _error;
  bool get hasRestaurants => _restaurants.isNotEmpty;

  //loading all restaurants from service
  Future<void> loadRestaurants() async {
    _setLoading(true);
  _clearError();

  try {
    //calling service to get restaurant data
    _restaurants = await _restaurantService.getRestaurants();

    // tell all listening widgets to rebuild
    notifyListeners();
  } catch (e) {
    //save the error if something goes wrong
    _setError(e.toString());
  } finally {
    //always stop loading whether success or failure (error)
    _setLoading(false);
  }
}

// select the restaurant when tapped on by the user
void selectRestaurant(Restaurant restaurant) {
  _selectedRestaurant = restaurant;
  notifyListeners();
}
```

How the loading flow works:

1. **Screen calls** provider.loadRestaurants()
2. **Loading starts** → _isLoading = true → UI shows spinner
3. **Service fetches data** → Either from API or mock data
4. **Success:** Data stored in _restaurants, UI rebuilds with restaurant list
5. **Error:** Error message stored, UI shows error screen with retry button

PizzaProvider

```
import 'package:flutter/foundation.dart';
import '../models/pizza.dart';
import '../models/allergen.dart';
import '../services/pizza_service.dart';

class PizzaProvider with ChangeNotifier {
  final PizzaService _pizzaService = PizzaService();

  List<Pizza> _pizzas = [];
  List<Pizza> _filteredPizzas = [];
  List<Allergen> _excludedAllergens = [];
  bool _isLoading = false;
  String? _error;
  int? _currentRestaurantId;

  //getters
  List<Pizza> get pizzas => _filteredPizzas;
  List<Pizza> get allPizzas => _pizzas;
  List<Allergen> get excludedAllergens => _excludedAllergens;
  bool get isLoading => _isLoading;
  String? get error => _error;
  bool get hasPizzas => _filteredPizzas.isNotEmpty;
  bool get hasAllergenFilter => _excludedAllergens.isNotEmpty;
  int? get currentRestaurantId => _currentRestaurantId;

  //load pizzas for a restaurant

  Future<void> loadPizzasByRestaurant(int restaurantId) async {
    _currentRestaurantId = restaurantId;
    _setLoading(true);
    _clearError();

    try {
      _pizzas = await _pizzaService.getPizzasByRestaurant(
        restaurantId,
        excludedAllergens:
            _excludedAllergens.isNotEmpty ? _excludedAllergens : null,
      );
      _applyAllergenFilter();
      notifyListeners();
    } catch (e) {
      _setError(e.toString());
    } finally {
      _setLoading(false);
    }
  }

  void toggleAllergenExclusion(Allergen allergen) {
    if (_excludedAllergens.contains(allergen)) {
      _excludedAllergens.remove(allergen);
    } else {
      _excludedAllergens.add(allergen);
    }

    if (_currentRestaurantId != null) {
      loadPizzasByRestaurant(_currentRestaurantId!);
    }
    notifyListeners();
  }
```

How the filtering system works step-by-step:

1. **User taps allergen filter** → toggleAllergenExclusion() called
2. **Update exclusion list** → Add/remove allergen from _excludedAllergens
3. **Reload pizza data** → Call service with new exclusion list
4. **Service filters data** → Returns only pizzas without excluded allergens
5. **Apply local filter** → Double-check filtering on frontend
6. **Notify listeners** → UI rebuilds with filtered pizza list

CartProvider - Handling Shopping Logic

```
import 'package:flutter/foundation.dart';
import '../models/cart_item.dart';
import '../models/pizza.dart';
import '../constants/api_constants.dart';

class CartProvider with ChangeNotifier {
  //private list of cart items
  final List<CartItem> _items = [];

  //getters
  List<CartItem> get items => List.unmodifiable(_items);
  int get itemCount => _items.length;
  int get totalQuantity => _items.fold(0, (sum, item) => sum + item.quantity);
  double get totalPrice =>
      _items.fold(0.0, (sum, item) => sum + item.totalPrice);
  bool get isEmpty => _items.isEmpty;
  bool get isNotEmpty => _items.isNotEmpty;

  //add pizza to cart or inc quantity
  void addPizza(Pizza pizza, {int quantity = 1}) {
    final existingIndex =
        _items.indexWhere((item) => item.pizza.id == pizza.id);
    if (existingIndex >= 0) {
      // increase quantity
      _items[existingIndex].quantity += quantity;
    } else {
      _items.add(CartItem(pizza: pizza, quantity: quantity));
    }
    notifyListeners();
  }

  void removePizza(int pizzaId) {
    _items.removeWhere((item) => item.pizza.id == pizzaId);
    notifyListeners();
  }

  void updateQuantity(int pizzaId, int quantity) {
    if (quantity <= 0) {
      removePizza(pizzaId);
      return;
    }
    final index = _items.indexWhere((item) => item.pizza.id == pizzaId);
    if (index >= 0) {
      _items[index].quantity = quantity;
      notifyListeners();
    }
  }
```

How cart operations work:

1. **User taps "Add to Cart"** → addPizza() called
2. **Check for existing item** → Search cart by pizza ID
3. **Smart adding:** Either increment quantity or add new item
4. **Automatic calculations** → Totals recalculate immediately
5. **UI updates** → Cart badge, totals, and quantity controls update everywhere

---

AllergenProvider - Managing Allergen Data

```
import 'package:flutter/foundation.dart';
import '../models/allergen.dart';
import '../services/allergen_service.dart';

class AllergenProvider with ChangeNotifier {
  final AllergenService _allergenService = AllergenService();

  List<Allergen> _allergens = [];
  bool _isLoading = false;
  String? _error;
```

```
//getters
List<Allergen> get allergens => _allergens;
bool get isLoading => _isLoading;
String? get error => _error;
bool get hasAllergens => _allergens.isNotEmpty;

//Load all allergens (caching the result)

Future<void> loadAllergens() async {
  if (_allergens.isNotEmpty) return;

  _setLoading(true);
  _clearError();

  try {
    _allergens = await _allergenService.getAllergens();
    notifyListeners();
  } catch (e) {
    _setError(e.toString());
  } finally {
    _setLoading(false);
  }
}
```

How the caching strategy works:

1. **First call** → Loads data from service, stores in _allergens
2. **Subsequent calls** → Returns immediately without API call
3. **Memory efficient** → Data loaded once and reused throughout app
4. **Refresh capability** → Can force reload if needed

AuthProvider - Managing User Sessions

```
import 'package:flutter/foundation.dart';
import '../models/user.dart';
import '../models/auth_response.dart';
import '../services/auth_service.dart';

class AuthProvider with ChangeNotifier {
  final AuthService _authService = AuthService();

  User? _currentUser;
  bool _isLoading = false;
  String? _error;
  bool _isLoggedIn = false;

  // Getters
  User? get currentUser => _currentUser;
  bool get isLoading => _isLoading;
  String? get error => _error;
  bool get isLoggedIn => _isLoggedIn;
  bool get isOwner => _currentUser?.isOwner ?? false;
  bool get isCustomer => _currentUser?.isCustomer ?? false;
  int? get ownerRestaurantId => _currentUser?.restaurantId;

  // Initialize auth state
  Future<void> initializeAuth() async {
    _setLoading(true);
    try {
      _currentUser = await _authService.getCurrentUser();
      _isLoggedIn = _currentUser != null;
      notifyListeners();
    } catch (e) {
      _setError('Failed to initialize authentication');
    } finally {
      _setLoading(false);
    }
  }

  // Login
```

```
Future<bool> login(String email, String password) async {
  _setLoading(true);
  _clearError();

  try {
    final authResponse = await _authService.login(email, password);
    _currentUser = authResponse.user;
    _isLoggedIn = true;

    notifyListeners();
    return true;
  } catch (e) {
    _setError(e.toString().replaceAll('Exception: ', ''));
    return false;
  } finally {
    _setLoading(false);
  }
}
```

How authentication flow works:

1. **User enters credentials** → Login form calls provider.login()
2. **Loading state** → UI shows loading spinner
3. **Service call** → AuthService attempts login with backend
4. **Success:** User data stored, _isLoggedIn = true, navigate to dashboard
5. **Failure:** Error message shown, user stays on login screen
6. **Role-based routing** → Different screens based on user type

---

How Providers Work Together

In main.dart, I register all providers at the app root level:

```
Widget build(BuildContext context) {
  return MultiProvider(
   providers: [
    ChangeNotifierProvider(create: (_) => RestaurantProvider()),
    ChangeNotifierProvider(create: (_) => PizzaProvider()),
    ChangeNotifierProvider(create: (_) => AllergenProvider()),
    ChangeNotifierProvider(create: (_) => CartProvider()),
    ChangeNotifierProvider(create: (_) => AuthProvider()),
   ],
```

Why at the root level:

- **Global access:** Any screen can access any provider when needed
- **Single instance:** Each provider exists once and maintains state throughout app lifecycle
- **Clean disposal:** Flutter automatically cleans up providers when app closes

Sometimes one screen needs data from multiple providers. Here's how I handle that:

```
Consumer2<AllergenProvider, PizzaProvider>(
       builder: (context, allergenProvider, pizzaProvider, child) {
        if (allergenProvider.isLoading) {
         return const Text('Loading allergens...');
        }

        return Wrap(
         spacing: 8,
         runSpacing: 8,
         children: allergenProvider.allergens.map((allergen) {
          final isExcluded =
            pizzaProvider.excludedAllergens.contains(allergen);
```

```
                  return FilterChip(
                    label: Text(allergen.name),
                    selected: isExcluded,
                    onSelected: (selected) {
                      pizzaProvider.toggleAllergenExclusion(allergen);
                    },
                    selectedColor: Colors.red.withValues(alpha: 0.2),
                    checkmarkColor: const Color.fromARGB(255, 200, 45, 45),
                    backgroundColor: Colors.grey[100],
                  );
                }).toList(),
              );
            },
          ),
```

What's happening here:

- **Consumer2**: Listens to both AllergenProvider and PizzaProvider
- **Automatic rebuilds**: UI updates when either provider changes
- **Cross-provider logic**: Filter chips show allergen names (from AllergenProvider) but track selection state (in PizzaProvider)

Navigation Architecture

Navigation Strategy

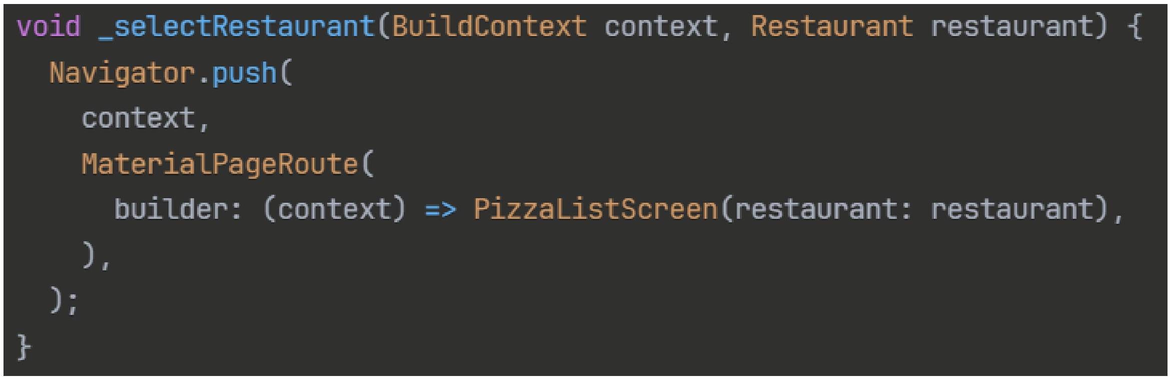
Simple Push/Pop Navigation

I used Flutter's built-in Navigator.push() and Navigator.pop() instead of complex routing packages.

Why this choice:

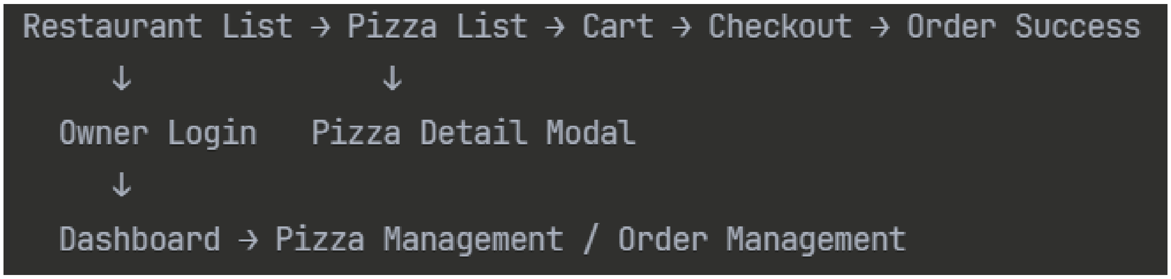
- **Easy to understand:** Clear cause and effect in navigation

- **Perfect for app size:** No need for complex routing setup
- **Direct control:** I can see exactly what triggers each navigation



```dart
void _selectRestaurant(BuildContext context, Restaurant restaurant) {
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => PizzaListScreen(restaurant: restaurant),
    ),
  );
}
```

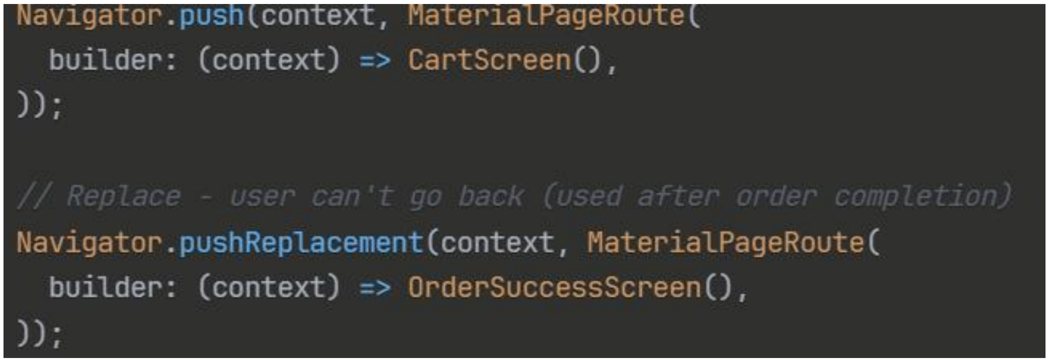
App Flow:



```
Restaurant List → Pizza List → Cart → Checkout → Order Success
     ↓                ↓
   Owner Login    Pizza Detail Modal
     ↓
   Dashboard → Pizza Management / Order Management
```

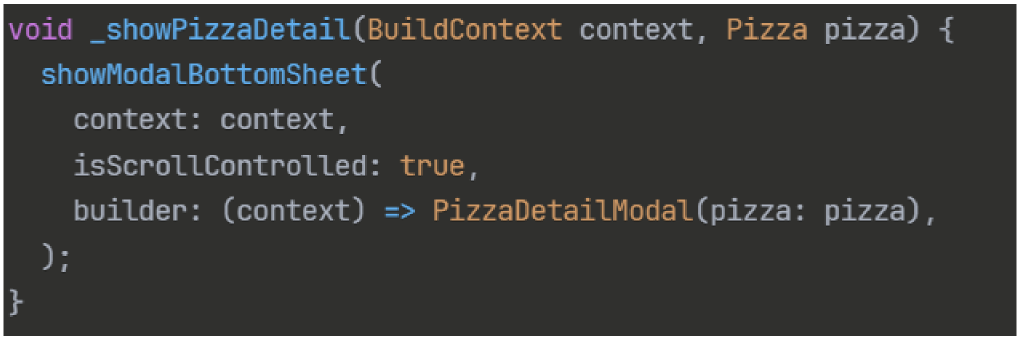
Two Key Navigation Patterns

Regular Screen Navigation:



```dart
Navigator.push(context, MaterialPageRoute(
  builder: (context) => CartScreen(),
));

// Replace - user can't go back (used after order completion)
Navigator.pushReplacement(context, MaterialPageRoute(
  builder: (context) => OrderSuccessScreen(),
));
```

Modal Bottom Sheets (for details)



```
void _showPizzaDetail(BuildContext context, Pizza pizza) {
  showModalBottomSheet(
    context: context,
    isScrollControlled: true,
    builder: (context) => PizzaDetailModal(pizza: pizza),
  );
}
```

Why modals work well:

- Users stay in context (can see pizza list behind modal)
- No back button confusion
- Easy to dismiss

Role-Based Navigation

How Different Users See Different Screens

Customer Flow: Restaurant List → Pizza List → Cart → Checkout
Owner Flow: Restaurant List (Press login button) → Login → Pizza/Order Management

Another key benefit of this navigation approach I took is that navigation and state work together automatically, hence, no manual coordination needed.

Data Layer & Services

Data Strategy

What: Mock-First with API Fallback

I built the app to work with mock data first.

Why this approach:

- **Faster development:** Could build features without waiting for backend
- **Easy testing:** Controlled, predictable data for testing

Service Layer Organization

What: One Service Per Data Type

Service Structure:

- **RestaurantService** → Restaurant data
- **PizzaService** → Pizza data and CRUD operations
- **AllergenService** → Allergen information
- **OrderService** → Order management
- **AuthService** → User authentication

Benefits:

- **Single responsibility:** Each service handles one data type
- **Easy to test:** Mock individual services independently
- **Clear structure:** Always know where to find data operations

---

Data Models

What: JSON Serialization with Code Generation

```
import 'package:json_annotation/json_annotation.dart';
import 'allergen.dart';

part 'pizza.g.dart';

@JsonSerializable()
class Pizza {
  final int id;
  final String name;
  final String description;
  final double price;
  final int restaurant;
  final String? image;
  final List<Allergen> allergens;

  Pizza(
      {required this.id,
      required this.name,
      required this.description,
      required this.price,
      required this.restaurant,
      this.image,
      required this.allergens});

  factory Pizza.fromJson(Map<String, dynamic> json) =>
_$PizzaFromJson(json);

  Map<String, dynamic> toJson() => _$PizzaToJson(this);

  //helper method to check if pizza contains any excluded allergens
  bool containsAllergens(List<Allergen> excludedAllergens) {
    return allergens.any((allergen) => excludedAllergens.contains(allergen));
  }

  // method to get allergen names as comma-separated string
  String get allergenNames {
    if (allergens.isEmpty) return 'No known allergens';
    return allergens.map((a) => a.name).join(', ');
  }

  //get formatted price
  String get formattedPrice {
    return '€${price.toStringAsFixed(2)}';
  }

  String? getImageUrl(String baseUrl) {
    if (image == null) return null;
    return '$baseUrl/files/$image';
  }
}
```

Why code generation:

- **Type safety:** Compiler catches JSON errors
- **Less boilerplate:** Auto-generated serialization code
- **Null safety:** Handles optional fields properly

UI/UX Design Patterns

Design System

I used Material Design as the foundation but customized colors to fit the pizza/food theme.

Why this approach:

- **Familiar patterns:** Users know how Material Design works

- **Accessibility built-in:** Material components have good accessibility
- **Food-friendly colors:** Warm colors that make food look appetizing

Reusable Component Patterns

Every list item (restaurants, pizzas, orders) uses the same card pattern for consistency.

- **Visual consistency:** Same layout for all list items
- **User familiarity:** Users quickly understand interaction patterns
- **Easy maintenance:** Change one pattern, updates everywhere

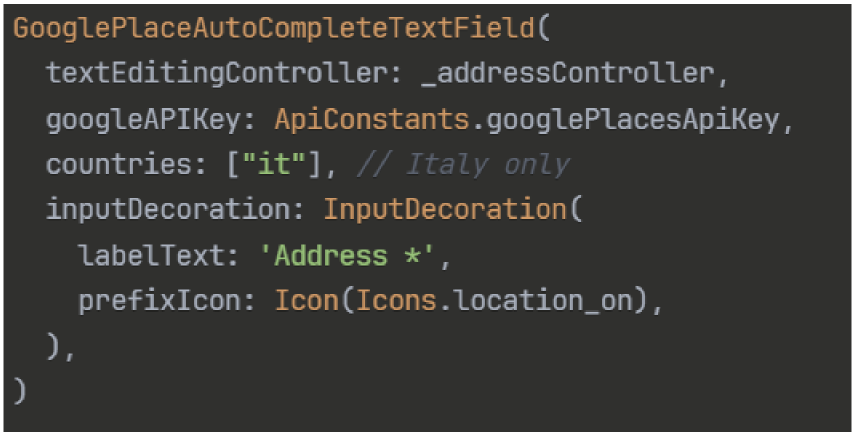
State-Driven UI Patterns

UI components automatically change based on app state without manual coordination. It was done by using Consumer pattern for Reactive UI.

Form Design Patterns

All forms use the same validation patterns and visual feedback.

Google Places Integration:



```
GooglePlaceAutoCompleteTextField(
  textEditingController: _addressController,
  googleAPIKey: ApiConstants.googlePlacesApiKey,
  countries: ["it"], // Italy only
  inputDecoration: InputDecoration(
    labelText: 'Address *',
    prefixIcon: Icon(Icons.location_on),
  ),
)
```