Case Study: Applying Clean Architecture in a Food Delivery App

Overview

Project: QuickBite – A scalable food delivery platform connecting users, restaurants, and delivery drivers.
Tech Stack: Kotlin (Android), Swift (iOS), Spring Boot (Backend), PostgreSQL (Database), React (Admin Dashboard)
Team Size: 12 (5 backend, 4 frontend/mobile, 3 QA/DevOps)
Duration: 9 months (from concept to MVP and first public release)

Challenge

QuickBite aimed to support rapid feature development (e.g., real-time tracking, promotions, multi-restaurant orders) while ensuring long-term maintainability. Early prototypes suffered from:

- Tightly coupled business logic with UI and database code.
- Difficulty testing core features without spinning up the full app.
- Slow onboarding of new developers due to unclear structure.
- High risk of regression bugs during updates.

The team decided to adopt **Clean Architecture** to address these issues.

Implementation of Clean Architecture

1. Layered Structure

The backend was organized into four distinct layers:

Layer	Responsibility
Domain	Business entities (e.g., Order, Restaurant) and use cases (e.g., PlaceOrderUseCase)
Application (Use Cases)	Application-specific logic and orchestration
Interface Adapters	Controllers, presenters, mappers; convert data between layers

Infrastructure	Database repositories, external APIs (payment, SMS), framework code

2. Dependency Rule Enforced

- Inner layers (Domain, Use Cases) had **no dependencies** on outer layers.
- Infrastructure implemented interfaces defined in the Use Cases layer (e.g., OrderRepository interface in Use Cases, implementation in Infrastructure).
- Dependency injection (via Dagger & Spring DI) ensured runtime binding.

3. Framework Independence

- Core logic was written in pure Kotlin/Java with no Spring annotations.
- Spring Boot was used only in the outermost layer for REST endpoints and DB access.
- This allowed unit testing of business rules without starting the server.

4. UI and Database Flexibility

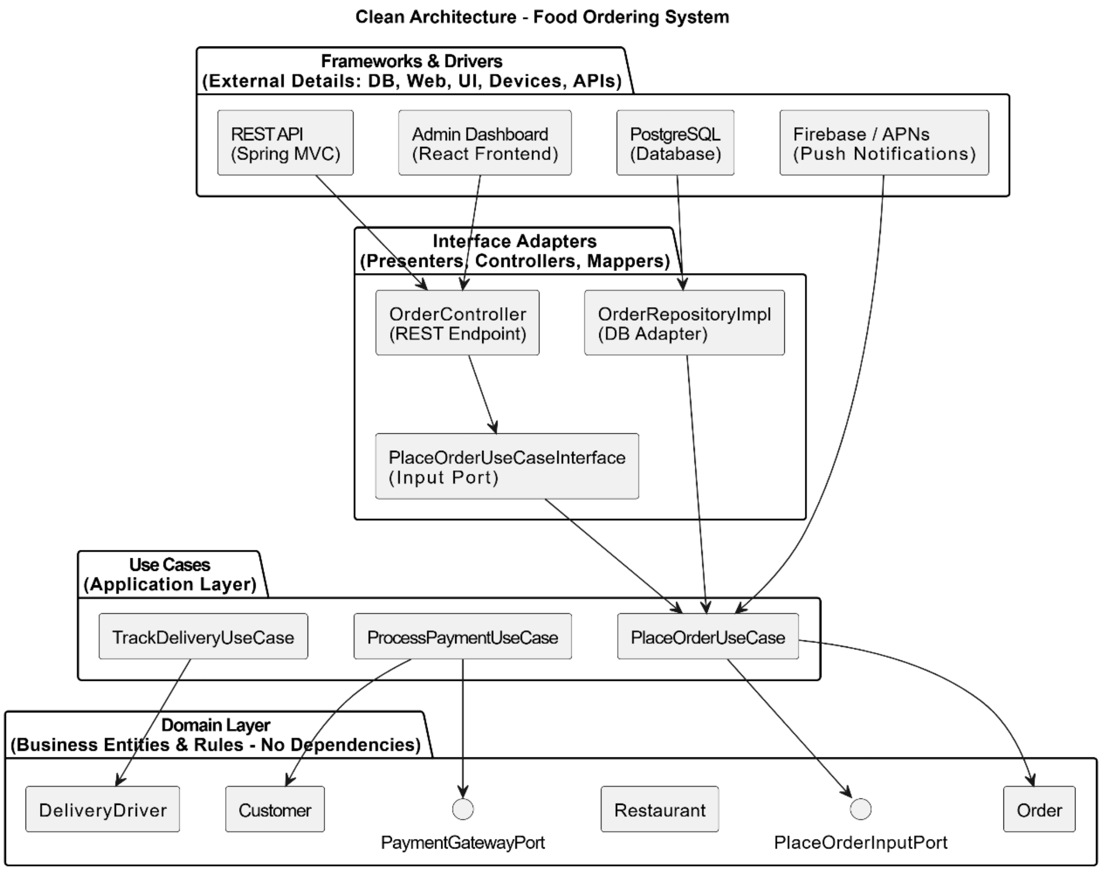
- The admin dashboard (React) and mobile apps communicated via clean API gateways.
- Initially used PostgreSQL; later migrated to CockroachDB for scalability — only infrastructure layer changed.
- UI redesigns did not impact order processing logic.

5. Testability

- Domain and Use Case layers achieved >90% unit test coverage.
- Integration tests focused only on adapters and infrastructure.
- CI pipeline ran domain tests in under 2 minutes, enabling fast feedback.

6. Business Logic Protection

- Core rules (e.g., "Orders can't be modified after payment") lived in the domain layer.
- All external components (APIs, UIs) had to go through use case interactors, enforcing consistency.


Clean Architecture - Food Ordering System
Frameworks & Drivers
(External Details: DB, Web, UI, Devices, APIs)
REST API (Spring MVC)
Admin Dashboard (React Frontend)
PostgreSQL (Database)
Firebase / APNs (Push Notifications)
Interface Adapters
(Presenters, Controllers, Mappers)
OrderController (REST Endpoint)
OrderRepositoryImpl (DB Adapter)
PlaceOrderUseCaseInterface (Input Port)
Use Cases
(Application Layer)
TrackDeliveryUseCase
ProcessPaymentUseCase
PlaceOrderUseCase
Domain Layer
(Business Entities & Rules - No Dependencies)
DeliveryDriver
Customer
PaymentGatewayPort
Restaurant
PlaceOrderInputPort
Order

Results

- ✓ **Faster Development**:
  New features (e.g., scheduled deliveries) were added 30% faster after architecture stabilization.
- ✓ **Improved Maintainability**:
  Bug reports dropped by 45% post-MVP due to clearer code ownership and fewer side effects.
- ✓ **Easier Onboarding**:
  New developers became productive within 2 weeks thanks to consistent structure and documentation.
- ✓ **Scalability**:
  Enabled seamless integration with third-party logistics APIs and multiple payment gateways.
- ✓ **Long-Term Flexibility**:
  Successfully pivoted from monolith to microservices (by splitting per Bounded Context) using the same architectural principles.

Lessons Learned

- **Upfront design pays off**: Initial setup took 3 extra weeks, but saved months in maintenance.

- **Enforce architecture with tools**: Used ArchUnit and custom linters to prevent dependency violations.
- **Training is key**: Team held workshops to align on patterns like dependency inversion and port/adapter flow.

Conclusion

By applying Clean Architecture, QuickBite built a resilient, scalable system centered around its business logic. The separation of concerns and strict dependency rules enabled agility, quality, and confidence in continuous delivery—proving that investing in solid architecture is essential for product longevity.