Senior Software Engineer Technical Test

# Overview

Spend **no more than 3 hours** in total. Solutions may be incomplete; we assess prioritisation, trade-offs, and code quality.

We’re looking for **elegant, maintainable, production-ready** code that demonstrates sound architecture and pragmatic decision-making aligned to a payments/ordering platform used by schools, parents, and students.

**Stack Focus:** .NET 8, C#, REST APIs, EF Core, automated testing (unit + integration)

# What to Submit

A small solution (ZIP or public repo) with:

* Source code (no binaries)
* Automated tests (unit + integration)
* A short **README.md** explaining:
  + Design/architecture decisions
  + Trade-offs and what you’d do next with more time
  + How to run locally (including tests)
* Optional: an **architecture sketch** (image/markdown) for Part 2.

Use any sensible structure; ensure it builds and runs locally.

# Part 1 — Build a Minimal Ordering & Payments API

Flexischools connects **parents** to **students** and lets them order from a **school canteen** (supplier). Implement a minimal slice of this domain and expose REST endpoints.

Domain:

Model the following:

* **Parent** – has an email, name, and a **wallet balance** (decimal, AUD).
* **Student** – has a name, is linked to exactly one Parent.
* **Canteen** – represents a school canteen; has **opening days** and an **order cut‑off time** per day (e.g., 9:30am).
* **MenuItem** – offered by a Canteen; has a price, an optional **daily stock** count, and optional **allergen tags** (e.g., nuts, dairy).
* **Order** – created by a Parent **for a Student** at a Canteen for **a fulfilment date** (typically the same day) and contains one or more MenuItems with quantities. Orders have states: *Placed*, *Fulfilled*, *Cancelled*.

Business Rules:

When creating an Order:

1. **Cut‑off** – Reject if created **after** the Canteen’s cut‑off for the given fulfilment date.
2. **Stock** – Reject if requested quantities exceed remaining daily stock for any MenuItem.
3. **Wallet balance** – Reject if Parent’s wallet balance is insufficient for the order total.
4. **Allergens (simple)** – If a Student has a recorded allergen (optional), any MenuItem carrying that allergen must **block** the order.
5. **Idempotency** – If the client provides an Idempotency-Key header, the create‑order operation must be idempotent for 24h (same request returns the same result without duplicating side effects).

On **successful** creation, set status to Confirmed, decrement stock, and **debit** the Parent’s wallet.

***Note:*** *You don’t need to integrate a real payment gateway; treat wallet as an account balance you can persist and update transactionally*

Persistence:

Use **EF Core** with any DB provider you like (SQLite/Postgres/SQL Server/InMemory). Please show you understand persistence concerns (transactions, concurrency) at a sensible level for the exercise.

***Seed data:*** *Initialise minimal data for one Parent, one Student (linked), one Canteen (with a cut‑off), and a few MenuItems during startup or via a migration/seeder. No endpoints are required to create these.*

**Required Endpoints:**

# Build two endpoints:

# ***POST /orders*** – Create an order (supports Idempotency-Key). Applies business rules: cut‑off, availability flag, wallet balance.

# ***GET /orders/{id}*** – Retrieve order status/details.

Swagger is encouraged for discoverability.

Non-functional Expectations:

* **Error handling** – Clear validation failures with meaningful messages.
* **Logging** – Include structured logs around order creation (inputs, totals, decisions). If you typically add correlation IDs, feel free to include them.
* **Configuration** – Reasonable defaults and separation of concerns (e.g., cut‑off times, DB connection strings).
* **Security (lightweight)** – It’s fine to stub auth; however, design your endpoints such that a future auth layer could authorise Parents to their Students.
* **CQRS** – Commands/queries split using MediatR (or similar), with thin controllers and rich domain.

Testing Focus:

* **Unit tests** – Domain/business logic (e.g., cut‑off, stock checks, wallet debit, idempotency behaviour).
* **Integration tests** – API + persistence path (happy path order creation, rejection scenarios). Spin up the chosen DB provider for real reads/writes (SQLite in‑memory acceptable).

Use **NUnit** + **FluentAssertions** (preferred) or your usual stack. Show how you keep tests fast, readable, and deterministic.

Stretch Goals (optional):

Pick one or two if time allows (not required):

* **Optimistic concurrency** on stock and wallet updates; show how you avoid overselling under contention.
* **Outbox pattern** (sketch/code) to emit an *OrderConfirmed* event for downstream systems (e.g., POS). Stub a background dispatcher.
* **Observability** – Expose a simple health/metrics endpoint and add log contexts (e.g., OrderId, ParentId).

# Part 2 — Architecture Sketch

Provide a brief diagram + notes for a production-grade version of this slice in the Flexischools ecosystem. Include:

* **API layer** (BFF or service), **Domain**, **Persistence**, optional **Outbox/Message Bus**, and **POS/Canteen integration** boundary.
* Where you’d enforce **idempotency**, **validation**, and **consistency**.
* How you’d handle **cut-off configuration**, **time‑zone handling** (Sydney/Australia focus), and **rollbacks/refunds**.

A diagram tool of your choice is fine. Hand-drawn and photographed is also acceptable.