

Concurrence et transactions

Deux clients, un seul plat

Imaginez. LivrExpress tourne en production. C'est vendredi soir. Le restaurant "Chez Mario" n'a plus qu'une seule pizza Margherita en stock. Deux clients, Alice et Bob, ont la page du restaurant ouverte. Ils voient tous les deux "1 en stock". Alice clique "Ajouter au panier". Bob clique une seconde après.

Résultat : les deux ont la pizza dans leur panier. Les deux passent commande. Les deux payent. Mais il n'y a qu'une seule pizza.

Bienvenue dans le monde de la **concurrence**.

On a construit nos repositories dans le chapitre 06. Ils lisent et écrivent proprement. Mais ils ne gèrent pas ce qui se passe quand plusieurs clients font la même action en même temps. C'est ce qu'on va voir ici.

Les deux problèmes fondamentaux

La mise à jour perdue (lost update)

Le restaurateur ouvre la fiche d'un plat pour corriger le prix : 12€ → 14€. Pendant ce temps, son associé ouvre la même fiche pour corriger la description. L'associé sauvegarde en premier. Puis le restaurateur sauvegarde.

Le problème : le restaurateur a écrasé la nouvelle description avec l'ancienne, parce qu'il travaillait sur une version obsolète. La modification de l'associé est **perdue**.

La lecture incohérente (inconsistent read)

Un gérant calcule le chiffre d'affaires de ses deux restaurants. Il lit : Restaurant 1 = 500€. Pendant ce temps, une commande de 50€ arrive sur le restaurant 1 et une de 30€ sur le restaurant 2. Il lit ensuite : Restaurant 2 = 330€. Total = 830€.

Le problème : 830€ n'a **jamais** été le bon total. Avant les commandes c'était 800€, après c'est 880€. Le gérant a lu des données **incohérentes** entre deux moments différents.

Deux stratégies : optimiste ou pessimiste

Le verrouillage optimiste : on détecte les conflits

On laisse tout le monde travailler librement, et on vérifie **au moment de sauvegarder** qu'il n'y a pas eu de conflit. L'astuce : ajouter un champ **version** à la table. Chaque modification incrémente la version. Si la version a changé entre le moment où on a lu et le moment où on sauvegarde, quelqu'un d'autre est passé avant nous.

```
# data/repositories/dish_repository.py
from data.database import get_database_connection
```

```

from domain.exceptions.order_exceptions import ConcurrencyConflictError

class DishRepository:

    def __init__(self):
        self.db = get_database_connection()

    def find_by_id(self, dish_id: int):
        row = self.db.execute(
            "SELECT id, name, price, description, version FROM dishes WHERE
id = :id",
            {"id": dish_id}
        ).fetchone()
        return Dish(
            id=row["id"],
            name=row["name"],
            price=row["price"],
            description=row["description"],
            version=row["version"]
        )

    def save(self, dish):
        result = self.db.execute(
            """UPDATE dishes
            SET name = :name, price = :price, description =
:description,
                version = version + 1
            WHERE id = :id AND version = :version""",
            {
                "name": dish.name,
                "price": dish.price,
                "description": dish.description,
                "id": dish.id,
                "version": dish.version
            }
        )
        if result.rowcount == 0:
            raise ConcurrencyConflictError(
                "Ce plat a été modifié par quelqu'un d'autre. Rechargez et
réessayez."
            )

```

La clause `WHERE id = :id AND version = :version` est la clé. Si la version a changé, `rowcount` vaut 0, personne n'a été mis à jour, et on lève une exception.

Le verrouillage pessimiste : on empêche les conflits

On verrouille la donnée **dès qu'on commence à travailler dessus**. Les autres doivent attendre.

```
def find_for_update(self, dish_id: int):
    # SELECT ... FOR UPDATE pose un verrou exclusif sur la ligne
    # Les autres requêtes qui veulent modifier ce plat devront attendre
    row = self.db.execute(
        "SELECT id, name, price, description FROM dishes WHERE id = :id FOR
UPDATE",
        {"id": dish_id}
    ).fetchone()
    return Dish(
        id=row["id"],
        name=row["name"],
        price=row["price"],
        description=row["description"]
    )
```

Comment choisir ?

- **Conflits rares** → verrouillage **optimiste**. C'est le cas pour la plupart des modifications dans LivrExpress (deux restaurateurs modifient rarement le même plat en même temps).
- **Conflits fréquents ou ressource limitée** → verrouillage **pessimiste**. C'est le cas du stock de pizzas : plusieurs clients peuvent tenter de réserver le même article simultanément.

Les deadlocks

Le deadlock, c'est quand deux opérations se bloquent mutuellement en attendant chacune que l'autre libère son verrou.

```
Alice : verrouille Restaurant → veut verrouiller Dish → attend Bob...
Bob   : verrouille Dish       → veut verrouiller Restaurant → attend
Alice...
```

↑ _____ |
DEADLOCK, personne n'avance

Solutions :

- **Détection automatique** : la base de données détecte le cycle et annule une des transactions
- **Timeout** : si une transaction attend trop longtemps, elle s'annule
- **Prévention** : toujours verrouiller les ressources dans le même ordre (toujours Restaurant avant Dish, jamais l'inverse)

Les transactions : ACID

Une **transaction** est un groupe d'opérations qui s'exécutent comme si c'était une seule opération indivisible. Les transactions garantissent quatre propriétés, résumées par l'acronyme ACID.

Atomicité, tout ou rien. Si une étape échoue, tout est annulé.

```
# Dans OrderService.mark_order_as_delivered
from data.database import get_database_connection

def mark_order_as_delivered(self, order_id: int) -> OrderWithStatus:
    db = get_database_connection()
    db.execute("BEGIN TRANSACTION")
    try:
        order = self.order_repo.find_by_id(order_id)

        if order is None:
            raise OrderNotFoundError(order_id)
        if order.delivered_at is not None:
            raise OrderAlreadyDeliveredError(order_id)

        order.delivered_at = datetime.now()
        self.order_repo.save(order)

        # Si on avait aussi un système de points fidélité à créditer :
        # self.loyalty_repo.add_points(order.customer_id, points)
        # → Si ça plante ici, la livraison est aussi annulée. Cohérent.

        db.execute("COMMIT")
        return self._build_order_with_status(order)
    except Exception:
        db.execute("ROLLBACK")
        raise
```

⚠ **Point de vigilance** : pour que ce **ROLLBACK** annule aussi ce que `order_repo.save()` a fait, le service et le repository doivent utiliser **la même connexion SQLite**. Si chacun ouvre sa propre connexion via `get_database_connection()`, le rollback du service n'annule que ses propres opérations. En pratique, on injecte la connexion dans le repository au lieu de la créer dedans, on en parlera quand on abordera l'injection de dépendances.

Cohérence, l'état de la base est valide avant et après la transaction. Les contraintes (clés étrangères, colonnes NOT NULL) sont respectées.

Isolation, les modifications d'une transaction en cours ne sont pas visibles par les autres avant le COMMIT.

Durabilité, une fois committée, la transaction est permanente, même en cas de panne serveur.

La règle d'or : une transaction par requête HTTP

Chaque requête HTTP doit ouvrir une transaction au début et la committer (ou la rollbacker) à la fin. Avec FastAPI :

```
# presentation/routes.py
@router.post("/orders/{order_id}/delivered", response_class=HTMLResponse)
async def mark_as_delivered(request: Request, order_id: int):
```

```

"""
Une requête HTTP = une transaction.
Si quelque chose plante, tout est annulé.
"""
try:
    order = order_service.mark_order_as_delivered(order_id)
    return templates.TemplateResponse(
        request,
        "order_row.html",
        {"order": order}
    )
except OrderNotFoundError:
    return HTMLResponse("Commande introuvable", status_code=404)
except OrderAlreadyDeliveredError:
    return HTMLResponse("Commande déjà livrée", status_code=409)

```

La présentation attrape les exceptions métier et retourne la bonne réponse HTTP. Le service gère la transaction. Chaque couche fait son job.

Les niveaux d'isolation

Tous les problèmes de concurrence ne nécessitent pas le même niveau de protection. Plus l'isolation est forte, plus les performances baissent.

Niveau	Problèmes possibles	Usage dans LivrExpress
Read Uncommitted	Lectures sales	Jamais
Read Committed	Lectures non répétables	Lecture simple (défaut SQLite)
Repeatable Read	Nouvelles lignes visibles	Calcul de CA
Serializable	Rien	Opérations financières

```

# Pour un calcul critique (chiffre d'affaires du restaurateur)
db.execute("BEGIN TRANSACTION")
db.execute("PRAGMA read_uncommitted = 0") # SQLite : isolation maximale
revenue = db.execute(
    """SELECT SUM(d.price)
    FROM orders o
    JOIN order_dishes od ON od.order_id = o.id
    JOIN dishes d ON d.id = od.dish_id
    WHERE o.restaurant_id = :id
    AND o.created_at >= :from_date""",
    {"id": restaurant_id, "from_date": today}
).fetchone()[0]
db.execute("COMMIT")

```

Transaction métier vs. transaction système

Une **transaction système**, c'est une transaction SQL : elle dure le temps d'une requête HTTP.

Une **transaction métier**, c'est un processus qui s'étale sur plusieurs étapes et plusieurs requêtes HTTP.
Pour LivrExpress : sélectionner les plats → entrer l'adresse → valider le paiement → confirmer.

La règle : **accumuler l'état côté client ou en session, écrire tout en base dans une seule transaction à la dernière étape.**

```
# Étape 3 : le client valide la commande, on écrit tout en une transaction
@router.post("/checkout/confirm", response_class=HTMLResponse)
async def confirm_checkout(request: Request, checkout_data: CheckoutData):
    """
    Toute la commande arrive en une seule requête.
    On écrit tout en une seule transaction atomique.
    """
    db = get_database_connection()
    db.execute("BEGIN TRANSACTION")
    try:
        # Créer la commande
        order_id = db.execute(
            """INSERT INTO orders (customer_id, restaurant_id,
delivery_address, created_at)
VALUES (:customer_id, :restaurant_id, :address, :now)""",
            {
                "customer_id": checkout_data.customer_id,
                "restaurant_id": checkout_data.restaurant_id,
                "address": checkout_data.delivery_address,
                "now": datetime.now().isoformat()
            }
        ).lastrowid

        # Associer les plats
        for dish_id in checkout_data.dish_ids:
            db.execute(
                "INSERT INTO order_dishes (order_id, dish_id) VALUES
(:order_id, :dish_id)",
                {"order_id": order_id, "dish_id": dish_id}
            )

        db.execute("COMMIT")
        return templates.TemplateResponse(request, "confirmation.html",
{"order_id": order_id})
    except Exception:
        db.execute("ROLLBACK")
        return HTMLResponse("Erreur lors de la commande", status_code=500)
```

Si le troisième INSERT plante, les deux premiers sont annulés. La base reste cohérente.



Les problèmes de concurrence ne sont pas que des bugs. Ce sont des **vulnérabilités exploitables**.

TOCTOU (Time-of-Check to Time-of-Use)

LivrExpress vérifie "ce client a-t-il assez de crédit ?" (check), puis débite (use). Entre les deux, une fraction de seconde. Un attaquant envoie 10 requêtes simultanées : les 10 checks passent (le crédit est encore là), puis les 10 débits s'exécutent. Le client a dépensé 10x son crédit.

La parade : faire le check et le use dans la **même opération atomique** :

```
UPDATE customers SET balance = balance - :amount
WHERE id = :id AND balance >= :amount
```

Si `rowcount == 0`, le client n'avait pas assez. Une seule requête, pas de fenêtre de vulnérabilité.

Exploitation du stock limité

L'exemple de la pizza en début de chapitre, c'est une vraie attaque. Un attaquant scripte des requêtes parallèles pour commander le même plat "dernière unité" plusieurs fois. Sans verrouillage pessimiste (`FOR UPDATE`) ou contrainte SQL, il les obtient toutes.

Bypass de rate limiting

Les protections "maximum 3 tentatives de mot de passe" sont souvent vulnérables. 100 requêtes simultanées peuvent toutes passer avant que le compteur ne soit incrémenté, si la vérification et l'incrémentation ne sont pas atomiques.

Double-spend dans les systèmes de paiement

Même crédit utilisé deux fois. Si LivrExpress vérifie le solde et débite dans des opérations séparées sans transaction, c'est exploitable.

Comment se protéger :

- Transactions avec le bon niveau d'isolation (`SERIALIZABLE` pour les opérations financières)
- Verrouillage pessimiste (`FOR UPDATE`) pour les ressources limitées (stock)
- Opérations atomiques : une seule requête SQL pour check + use

Pour les pentesters : les race conditions sont difficiles à trouver en test manuel. Utilisez Turbo Intruder (Burp Suite) pour envoyer des requêtes parallèles. Ciblez les fonctionnalités critiques : paiements, réservations, tout ce qui implique une ressource limitée ou un solde.