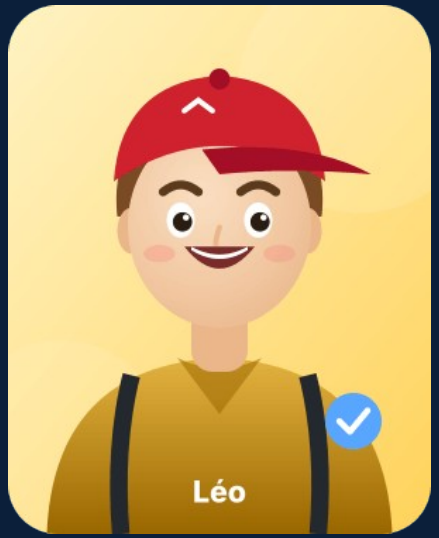
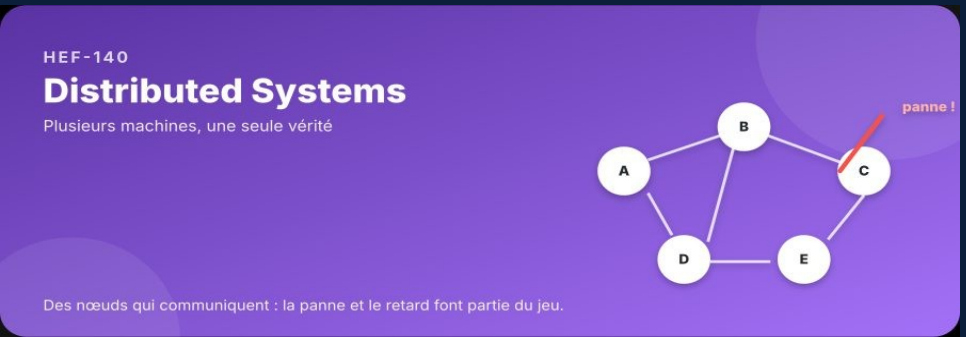
HEF-140 · MISSION 04 / 10

Idempotence



Si une opération peut être rejouée sans changer le résultat, les retries deviennent sûrs — et la plupart des problèmes de livraison disparaissent.



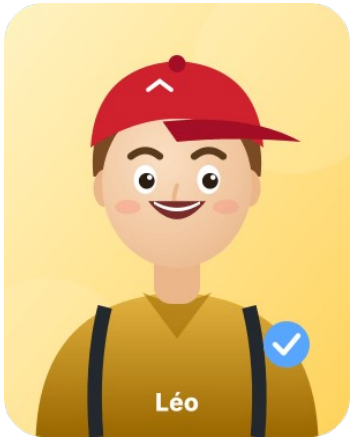
Badge : Rejoueur sans danger

105 min · avancé

L'histoire

Léo clique deux fois sur « Payer » parce que la page tarde. Son navigateur envoie deux requêtes. Le paiement passe deux fois. Un autre jour, le service de paiement répond lentement ; l'inscription relance ; la carte est débitée deux fois. Un troisième jour, un message de la file est livré en double après un redémarrage, et un cours est facturé deux fois à un formateur.

« Trois causes différentes, une seule cure, dit Sam : rendre l'opération idempotente. Une opération idempotente, tu peux la jouer une fois ou dix fois, l'état final est le même. Alors les doublons ne font plus peur. »



Tu apprends à reconnaître les opérations naturellement idempotentes, à créer des clés d'idempotence, à implémenter une déduplication atomique avec une base de données, et à gérer les cas difficiles : requêtes concurrentes, réutilisation de clé, panne entre l'effet et l'enregistrement.

Ta mission

À la fin, je serai capable de…

1. définir l'idempotence et distinguer opérations naturellement idempotentes ou non
2. implémenter une clé d'idempotence avec un stockage atomique (contrainte d'unicité)
3. gérer les requêtes concurrentes portant la même clé et la réutilisation d'une clé avec une autre charge
4. rendre un consommateur de messages idempotent en rendant atomique effet et enregistrement
5. décrire la durée de conservation des clés et ses compromis

Défi d'échauffement

Défi d'échauffement

Un interrupteur « allumer » est-il idempotent ? Et un bouton « changer l'état de la lampe » (bascule) ? Si tu appuies par erreur deux fois sur chacun, quel est l'état final ? Donne trois exemples d'actions de la vie quotidienne qui sont idempotentes, et trois qui ne le sont pas.

Le concept

Définition

Une opération `f` est **idempotente** si l'appliquer plusieurs fois donne le même état que l'appliquer une fois : `f(f(x)) = f(x)`.

Idempotente	Non idempotente
`PUT /membres/42 {nom: "Ada"}` (remplace)	`POST /paiements` (crée)
`DELETE /cours/7`	`incrementer(compteur)`
`solde = 60` (affectation)	`solde -= 40`
« marquer comme lu »	« envoyer un e-mail »
`INSERT ... ON CONFLICT DO NOTHING`	`INSERT` ordinaire

(Le *résultat* de `DELETE` répété peut changer — 200 puis 404 — mais l'**état** final est le même.)

Le concept

Pourquoi c'est vital en distribué

Les retries, les doublons de messages, les rejeux après panne sont inévitables (missions 2 et 3). **Au moins une fois + idempotence = effet exactement une fois.**

Rendre naturellement idempotent

Transformer une opération relative en opération absolue :

Au lieu de	Utiliser
`solde -= 40`	`solde = 60` (avec contrôle de version) ou un enregistrement `debit(id=…, 40)` unique
« ajouter l'élément à la liste »	ensemble (`set`) ou « ajouter s'il n'existe pas »
`compteur += 1`	compteur dérivé du nombre d'événements distincts

Le concept

La clé d'idempotence

Quand l'opération est intrinsèquement non idempotente (créer un paiement), le **client génère une clé unique** (UUID) pour **chaque intention** et la joint à la requête. Le serveur :

1. regarde si la clé a déjà été vue ;
2. **si oui** : renvoie la **réponse mémorisée** sans ré-exécuter ;
3. **si non** : exécute, enregistre résultat **et** clé, répond.

Règles :

- la clé identifie une **intention**, pas une tentative : toutes les relances d'une même intention portent la **même** clé ;
- **vérifier la charge** : la même clé avec un corps différent est une erreur (422/409) ;

Le concept

La clé d'idempotence (suite)

- **atomicité** : l'enregistrement de la clé et de l'effet doivent se faire **dans la même transaction**, sinon une panne entre les deux crée un doublon ou une perte ;
- **concurrence** : deux requêtes simultanées avec la même clé ne doivent pas toutes les deux exécuter : contrainte d'**unicité** en base ou verrou ;
- **durée de conservation** : assez longue pour couvrir les relances (24 h typiquement), assez courte pour limiter l'espace.

États d'une clé

```
inconnue → en_cours → terminée (avec réponse)
                   ↘ échec définitif (réponse d'erreur mémorisée, ou clé libérée si
    échec transitoire)
```

Une requête qui arrive pendant l'état `en_cours` reçoit **409 « en cours »** (ou attend).

Le concept

Le piège de la panne au mauvais moment

```
1. débiter la carte        ← l'effet est fait
2. [panne]
3. enregistrer la clé      ← jamais exécuté
→ au rejeu, la clé est inconnue : on débite de nouveau
```

Remèdes : (a) effet et clé dans **une transaction locale** quand le système de stockage est le même ; (b) quand l'effet est **externe** (API d'un prestataire), **transmettre la clé au prestataire** (qui déduplique de son côté) ; (c) sinon, un protocole de **réconciliation**.

Consommateurs de messages idempotents

Le concept

Consommateurs de messages idempotents (suite)

Une file livre **au moins une fois**. Le consommateur garde une table `messages_traités(id)` et, **dans la même transaction que l'effet**, insère l'identifiant : un message rejoué est ignoré. C'est le patron « boîte de réception » (*inbox*, mission 8).

Ordre et commutativité

L'idempotence ne suffit pas si les messages arrivent dans un **ordre** différent. Solutions : numéros de version (n'appliquer que si plus récent), opérations **commutatives** (ensembles, compteurs CRDT — mission 5).

🧠 ***Astuce** : pour chaque endpoint ou consommateur, demande : « que se passe-t-il si cette requête arrive deux fois, en même temps, ou après une panne ? » Si la réponse est « un problème », ajoute une clé.*

Exemple pas à pas

Problème : implémenter une API de paiement avec clé d'idempotence adossée à une base SQL (unicité atomique), gérer le doublon, la requête concurrente, la clé réutilisée avec une autre charge, et la panne entre effet et enregistrement ; puis rendre idempotent un consommateur de messages.

```
import hashlib, json, sqlite3, threading

class ErreurApi(Exception):
    def __init__(self, statut, message): super().__init__(f"{statut} {message}");
   self.statut = statut

class ServicePaiement:
    def __init__(self, panne_apres_effet=False):
        self.bd = sqlite3.connect(":memory:", check_same_thread=False, isolation_l
   evel=None)
        self.bd.executescript('''
            CREATE TABLE soldes (membre TEXT PRIMARY KEY, solde INTEGER);
            CREATE TABLE cles (cle TEXT PRIMARY KEY, empreinte TEXT, etat TEXT, re
   ponse TEXT);
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            CREATE TABLE paiements (id INTEGER PRIMARY KEY, membre TEXT, montant I
NTEGER, cle TEXT);
            INSERT INTO soldes VALUES ('ada', 100);
        ''')
        self.verrou, self.panne_apres_effet = threading.Lock(), panne_apres_effet

    def payer(self, cle, membre, montant):
        empreinte = hashlib.sha256(json.dumps([membre, montant]).encode()).hexdige
st()
        with self.verrou:                                                # séri
alise les accès à la connexion partagée
            self.bd.execute("BEGIN IMMEDIATE")
            try:
                ligne = self.bd.execute("SELECT empreinte, etat, reponse FROM cles
 WHERE cle = ?", (cle,)).fetchone()
                if ligne:
                    self.bd.execute("ROLLBACK")
                    if ligne[0] != empreinte: raise ErreurApi(422, "clé déjà utili
sée avec une autre requête")
```

Exemple pas à pas

Exemple pas à pas (suite)

```
                if ligne[1] == "en_cours": raise ErreurApi(409, "requête déjà
en cours")
                return json.loads(ligne[2]), True                       # DOUBL
ON : on renvoie la réponse mémorisée
            solde = self.bd.execute("SELECT solde FROM soldes WHERE membre = ?
", (membre,)).fetchone()[0]
            if solde < montant:
                reponse = {"statut": "refuse", "raison": "solde insuffisant"}
            else:
                self.bd.execute("UPDATE soldes SET solde = solde - ? WHERE mem
bre = ?", (montant, membre))
                self.bd.execute("INSERT INTO paiements (membre, montant, cle)
VALUES (?, ?, ?)", (membre, montant, cle))
                reponse = {"statut": "accepte", "solde": solde - montant}
            self.bd.execute("INSERT INTO cles VALUES (?, ?, 'termine', ?)", (c
le, empreinte, json.dumps(reponse)))
            if self.panne_apres_effet:
                self.panne_apres_effet = False
                raise RuntimeError("panne du serveur après l'effet, avant la r
```

Exemple pas à pas

Exemple pas à pas (suite)

```
   éponse")       # la transaction est ANNULÉE entièrement
                self.bd.execute("COMMIT")
                return reponse, False
            except RuntimeError:
                self.bd.execute("ROLLBACK"); raise
    def solde(self, membre): return self.bd.execute("SELECT solde FROM soldes WHER
   E membre = ?", (membre,)).fetchone()[0]
    def nb_paiements(self): return self.bd.execute("SELECT COUNT(*) FROM paiements
   ").fetchone()[0]

# ---- 1. un clic, deux clics, dix clics : même clé, un seul débit ----
s = ServicePaiement()
reponses = [s.payer("cle-abc", "ada", 40) for _ in range(5)]
print("réponses :", [(r["statut"], doublon) for r, doublon in reponses], "| solde
   :", s.solde("ada"))
assert s.solde("ada") == 60 and s.nb_paiements() == 1
assert [d for _, d in reponses] == [False, True, True, True, True] and all(r == re
   ponses[0][0] for r, _ in reponses)
```

Exemple pas à pas (suite)

```
# ---- 2. même clé, autre charge : refusé ----
try: s.payer("cle-abc", "ada", 99); raise AssertionError
except ErreurApi as e: print("clé réutilisée :", e); assert e.statut == 422

# ---- 3. deux requêtes SIMULTANÉES avec la même clé ----
s = ServicePaiement(); sorties, barriere = [], threading.Barrier(8)
def requete():
    barriere.wait()
    try: sorties.append(s.payer("cle-concurrente", "ada", 30))
    except ErreurApi as e: sorties.append(e)
ts = [threading.Thread(target=requete) for _ in range(8)]; [t.start() for t in ts]
   ; [t.join() for t in ts]
assert s.nb_paiements() == 1 and s.solde("ada") == 70                    # 8 req
   uêtes, UN seul débit
print("8 requêtes simultanées → paiements créés :", s.nb_paiements())

# ---- 4. panne après l'effet : l'effet ET la clé sont annulés ensemble → le rejeu
    réussit sans doublon ----
s = ServicePaiement(panne_apres_effet=True)
```

Exemple pas à pas (suite)

```
try: s.payer("cle-panne", "ada", 40); raise AssertionError
except RuntimeError as e: print("panne :", e)
assert s.solde("ada") == 100 and s.nb_paiements() == 0                    # rien
   n'a été appliqué (transaction atomique)
r, doublon = s.payer("cle-panne", "ada", 40)                              # le cl
   ient réessaie avec la MÊME clé
assert r["statut"] == "accepte" and not doublon and s.solde("ada") == 60 and s.nb_
   paiements() == 1
print("rejeu après panne : un seul débit")

# ---- 5. consommateur de messages idempotent ----
class Consommateur:
    def __init__(self):
        self.bd = sqlite3.connect(":memory:", isolation_level=None)
        self.bd.executescript("CREATE TABLE traites (id TEXT PRIMARY KEY); CREATE
   TABLE factures (cours TEXT, montant INTEGER);")
    def traiter(self, message):
        self.bd.execute("BEGIN")
        try:
```

Exemple pas à pas (suite)

```
            self.bd.execute("INSERT INTO traites VALUES (?)", (message["id"],))
         # lève IntegrityError si déjà vu
            self.bd.execute("INSERT INTO factures VALUES (?, ?)", (message["cours"
   ], message["montant"]))
            self.bd.execute("COMMIT"); return "traité"
        except sqlite3.IntegrityError:
            self.bd.execute("ROLLBACK"); return "doublon ignoré"
c = Consommateur()
livraisons = [{"id": "m1", "cours": "HEF-010", "montant": 40}, {"id": "m2", "cours
   ": "HEF-020", "montant": 30}, {"id": "m1", "cours": "HEF-010", "montant": 40},
   {"id": "m2", "cours": "HEF-020", "montant": 30}]
print([c.traiter(m) for m in livraisons])
assert c.bd.execute("SELECT COUNT(*), SUM(montant) FROM factures").fetchone() == (
   2, 70)
```

Lecture :

Exemple pas à pas (suite)

1. Cinq envois de la même intention (même clé) produisent **un** débit ; les réponses sont **identiques**, marquées comme doublons.
2. La même clé avec une **autre charge** (99 au lieu de 40) est refusée (422) : la clé identifie une intention précise.
3. Huit requêtes **simultanées** avec la même clé ne créent qu'**un** paiement (l'unicité est garantie par la base).
4. L'effet **et** l'enregistrement de la clé sont dans **la même transaction** : une panne les annule ensemble ; le rejeu avec la même clé s'exécute normalement, **sans doublon ni perte**.
5. Le consommateur de messages enregistre l'identifiant du message **dans la transaction de l'effet** : les doublons sont ignorés, le total facturé est correct (70).

À toi, avec aide

Un compteur de vues reçoit des événements `{"id": ..., "page": ...}` livrés **au moins une fois** (certains en double). La version actuelle fait `vues[page] += 1`.

1. Montre que le résultat est faux avec des doublons.
2. Réécris `enregistrer(evenement)` pour qu'il soit idempotent (avec une mémoire des identifiants vus, bornée par une fenêtre de 1000 identifiants récents — explique le risque de cette borne).
3. Propose une alternative qui ne nécessite aucune mémoire des identifiants (indice : conserver l'ensemble des identifiants par page ou utiliser une structure approximative). Quel est son coût ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise un `collections.OrderedDict` ou `deque` + `set` pour la fenêtre.

Indice 2

Si un doublon arrive après que son identifiant est sorti de la fenêtre, il sera recompté.

Indice 3

Un ensemble par page rend le comptage idempotent et commutatif.

À toi, avec aide · réponse

Réponse

```
from collections import OrderedDict, defaultdict

EVENEMENTS = [{"id": i % 7, "page": "/cours"} for i in range(10)] + [{"id": 100, "
   page": "/prix"}] * 3
naif = defaultdict(int)
for e in EVENEMENTS: naif[e["page"]] += 1
print("naïf :", dict(naif))                                  # compte les doublo
   ns

class CompteurFenetre:
    def __init__(self, fenetre=1000):
        self.vus, self.fenetre, self.vues = OrderedDict(), fenetre, defaultdict(in
   t)
    def enregistrer(self, e):
        if e["id"] in self.vus: return False
        self.vus[e["id"]] = True
        if len(self.vus) > self.fenetre: self.vus.popitem(last=False)     # oublie
    les plus anciens
        self.vues[e["page"]] += 1; return True
```

À toi, avec aide · réponse

Réponse (suite)

```
class CompteurEnsemble:                                          # idempotent ET co
   mmutatif, sans fenêtre : le coût est la mémoire (un identifiant par événement)
    def __init__(self): self.ids = defaultdict(set)
    def enregistrer(self, e): self.ids[e["page"]].add(e["id"])
    def vues(self, page): return len(self.ids[page])

f, a = CompteurFenetre(), CompteurEnsemble()
for e in EVENEMENTS:
    f.enregistrer(e); a.enregistrer(e)
assert dict(f.vues) == {"/cours": 7, "/prix": 1} and a.vues("/cours") == 7 and a.v
   ues("/prix") == 1
# doublon tardif hors fenêtre : la version à fenêtre se trompe
f2 = CompteurFenetre(fenetre=3)
for i in range(5): f2.enregistrer({"id": i, "page": "/x"})
f2.enregistrer({"id": 0, "page": "/x"})                           # doublon de l'év
   énement 0, déjà sorti de la fenêtre
assert f2.vues["/x"] == 6                                        # recompté !
print("fenêtre trop courte → recomptage ; ensemble complet → exact mais mémoire O(
```

Réponse (suite)

```
    n)")
```

La fenêtre borne la mémoire mais **n'est correcte que si les doublons arrivent dans la fenêtre**. L'ensemble par page est **exact et commutatif** (ordre indifférent) mais coûte une entrée par identifiant ; pour des milliards d'événements on utilise des structures approximatives (filtre de Bloom, HyperLogLog) en acceptant une petite erreur.

À toi, en solo

Saga de réservation idempotente. Une réservation enchaîne trois étapes sur trois services : `reserver_place(cours)`, `debiter(membre, montant)`, `envoyer_confirmation(membre)`. Chaque service peut échouer ou répondre en double.

1. donne à chaque étape sa propre clé dérivée de la clé de la réservation (`f"{cle}:place"`, etc.) ;
2. écris l'orchestrateur avec **reprise** : un journal enregistre l'étape atteinte ; si le processus plante au milieu, la relance reprend où on en était sans refaire ce qui est fait ;
3. écris les trois services avec leur propre déduplication ;
4. simule des pannes à chaque point (avant/après chaque appel, avant/après l'écriture du journal) avec une injection exhaustive : pour **chaque point de panne possible**, rejoue jusqu'au succès ; vérifie que le résultat final est toujours : 1 place, 1 débit, 1 confirmation.

À toi, en solo · réponse

Pistes de correction

```
class Plantage(Exception): pass

class Service:                                                  # chaque service
   déduplique par clé et compte ses effets
    def __init__(self, nom): self.nom, self.vus, self.effets = nom, {}, 0
    def appeler(self, cle, charge):
        if cle in self.vus: return self.vus[cle]
        self.effets += 1; self.vus[cle] = f"{self.nom}:ok"; return self.vus[cle]

class Orchestrateur:
    def __init__(self, services, journal, points_de_panne):
        self.services, self.journal, self.pannes = services, journal, points_de_pa
   nne      # journal : dictionnaire durable
        self.compteur_points = 0
    def point(self):                                            # point d'injecti
   on de panne
        self.compteur_points += 1
        if self.compteur_points in self.pannes: self.pannes.discard(self.compteur_
   points); raise Plantage(f"panne au point {self.compteur_points}")
```

Pistes de correction (suite)

```
    def executer(self, cle):
        for etape in ("place", "debit", "confirmation"):
            if self.journal.get((cle, etape)): continue          # déjà fait : on
   saute (reprise)
            self.point()                                          # panne AVANT l'
   appel
            self.services[etape].appeler(f"{cle}:{etape}", {})
            self.point()                                          # panne APRÈS l'
   appel, AVANT l'écriture du journal : l'effet existe, le journal l'ignore
            self.journal[(cle, etape)] = True
            self.point()                                          # panne APRÈS l'
   écriture du journal
        return "terminé"

def rejouer(points):
    services = {e: Service(e) for e in ("place", "debit", "confirmation")}
    journal, essais = {}, 0
    pannes = set(points)
    while True:
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        essais += 1
        orch = Orchestrateur(services, journal, pannes)
        try: orch.executer("res-1"); break
        except Plantage: continue
    return {e: s.effets for e, s in services.items()}, essais

# injection exhaustive : une panne à chaque point (9 points) puis toutes les paire
   s de pannes
attendu = {"place": 1, "debit": 1, "confirmation": 1}
for p in range(1, 10):
    effets, essais = rejouer({p}); assert effets == attendu, (p, effets)
for p in range(1, 10):
    for q in range(p + 1, 12):
        effets, essais = rejouer({p, q}); assert effets == attendu, (p, q, effets)
print("9 pannes simples et 63 paires de pannes : toujours 1 place, 1 débit, 1 conf
   irmation")
```

Pistes de correction (suite)

Deux ingrédients rendent la saga sûre malgré les pannes : le **journal** (reprise sans tout refaire) et la **déduplication par clé dérivée** dans chaque service (si la panne survient entre l'appel et l'écriture du journal, le rejeu de l'appel est sans effet supplémentaire). Le test **exhaustif** d'injection de panne est la bonne façon de s'en convaincre.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Clé d'idempotence générée à chaque tentative	Chaque relance est une nouvelle intention : doublons	Une clé par intention, réutilisée par les relances
Enregistrer la clé et l'effet séparément	Panne entre les deux : doublon ou perte	Même transaction, ou transmettre la clé au prestataire
Ne pas vérifier la charge associée à la clé	Une clé réutilisée par erreur renvoie une réponse sans rapport	Empreinte de la requête et erreur 422
Clés conservées trop peu de temps	Un rejeu tardif crée un doublon	Durée supérieure à la fenêtre de relance maximale

Mini-boss

Mini-boss

API de paiement de HashBox, version industrielle. Écris un service avec :

1. création de paiement avec clé d'idempotence (en-tête `Idempotency-Key`), empreinte de la requête, réponses mémorisées, expiration à 24 h (horloge simulée) ;
2. états `en_cours` / `termine` avec réponse 409 pour une requête concurrente ;
3. remboursement idempotent (un remboursement ne peut dépasser le paiement, même avec des clés différentes pour la même demande métier — précise comment tu le détectes) ;
4. consommateur de messages `paiement.confirme` avec boîte de réception atomique ;
5. test d'**injection exhaustive** de pannes (avant/après chaque écriture) et de **rejeux aléatoires** (500 graines : doublons, réordonnancements, pertes de réponse) avec invariants : somme des soldes conservée, un paiement par intention, jamais de solde négatif ;
6. rapport : tableau des scénarios de panne et de ce qui garantit chaque invariant.

Mini-boss · réponse

Pistes de correction

Un bon travail : tables `cles(cle, empreinte, etat, reponse, expire)`, `paiements`, `remboursements`, `traites` ; transaction unique pour effet + clé ; expiration vérifiée avec horloge injectée ; injection de panne par compteur de points ; rejeux aléatoires avec invariants vérifiés ; tableau où chaque invariant renvoie à un mécanisme (unicité SQL, transaction, empreinte, contraintes CHECK). La réponse au point 3 : une clé métier unique (par exemple `remboursement:{paiement_id}:{motif_id}`) ou une contrainte `SUM(rembourse) <= montant` appliquée dans la transaction.

Quiz éclair

Quiz éclair

1. Une opération idempotente est une opération qui :

A. donne le même état final qu'on l'applique une fois ou plusieurs fois
B. s'exécute toujours instantanément
C. ne peut pas échouer

2. Une clé d'idempotence doit être :

A. nouvelle à chaque tentative
B. la même pour toutes les relances d'une même intention
C. secrète et chiffrée

3. Pour qu'un consommateur de messages soit idempotent, on enregistre l'identifiant du message :
A. dans un journal séparé, plus tard

Quiz éclair (suite)

B. jamais

C. dans la même transaction que l'effet

4. 🔁 *Révision mission 03* : Avec 3 couches qui réessaient chacune 3 fois, une dépendance en panne totale reçoit :

A. 27 fois la charge normale

B. 3 fois la charge

C. aucune charge

5. 🔁 *Révision mission 03* : Un budget de retries sert à :

A. augmenter le nombre de relances

B. plafonner les relances à une fraction du trafic pour éviter l'avalanche

Quiz éclair

Quiz éclair (suite)

C. supprimer les délais d'attente

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

1. Quelle opération de ton système, rejouée deux fois, causerait un dommage réel ?
2. À quel endroit génères-tu aujourd'hui des identifiants : à chaque tentative, ou par intention ?

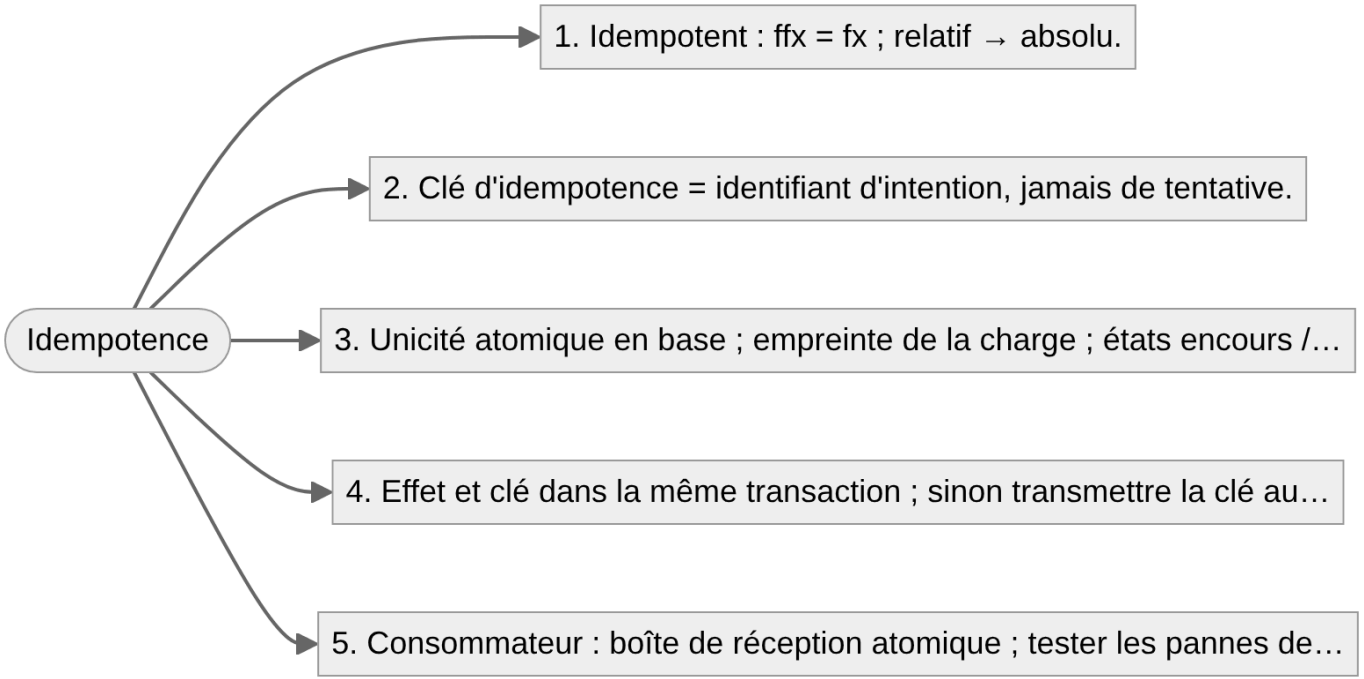
À retenir

À retenir

1. Idempotent : f(f(x)) = f(x) ; relatif → absolu.
2. Clé d'idempotence = identifiant d'intention, jamais de tentative.
3. Unicité atomique en base ; empreinte de la charge ; états en_cours / terminé.
4. Effet et clé dans la même transaction ; sinon transmettre la clé au prestataire.
5. Consommateur : boîte de réception atomique ; tester les pannes de façon exhaustive.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Rejoueur sans danger

Prochaine mission : Réplication

