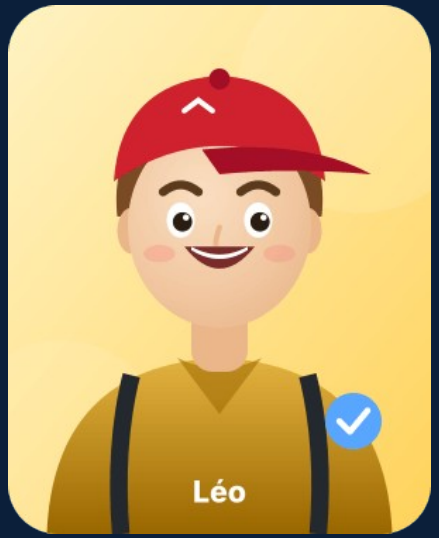
HEF-110 · MISSION 04 / 10

# Tests d'intégration




*Chaque pièce fonctionne seule. Mais la machine, elle, fonctionne-t-elle ? Les tests d'intégration vérifient les coutures.*



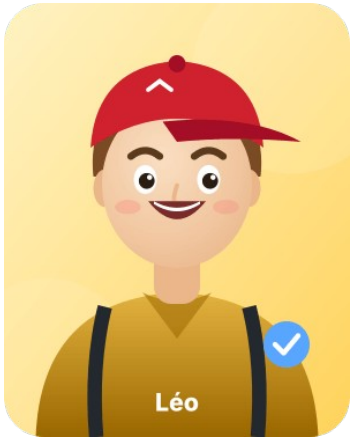

**Badge : Assembleur de preuves**

100 min · intermédiaire

L'histoire

# L'histoire

Toutes les pages de la suite verte. Pourtant, le jour du déploiement, l'inscription à un cours échoue pour tout le monde. Le coupable : le service d'inscription appelle le dépôt avec `(membre, cours)`, alors que le dépôt attend `(cours, membre)`. Les tests unitaires de chacun passaient : chacun testait avec **son** hypothèse sur l'autre.




> *« Les bugs adorent les coutures, dit Sam. Entre deux modules, entre le code et la base, entre ton service et celui d'un autre. Un test d'intégration prend les **vraies** pièces et les fait travailler ensemble. »*

**Tu apprends à tester des composants réels ensemble : base de données, fichiers, transactions, jeux de données maîtrisés.**

Ta mission

# À la fin, je serai capable de…

1. expliquer ce que testent les tests d'intégration et ce que les tests unitaires ne voient pas
2. tester du code avec une vraie base de données (SQLite en mémoire)
3. isoler chaque test grâce à des transactions ou une base neuve
4. gérer les données de test et les fichiers temporaires
5. décider quoi tester en unitaire et quoi en intégration

# Défi d'échauffement

Une équipe de relais 4×100 m : chaque coureur est excellent seul, mais le témoin tombe à chaque passage. Quels tests n'auraient **jamais** détecté le problème (chronométrer chaque coureur seul) ? Quel test le détecterait immédiatement ?

Le concept

# Ce qu'on teste

Un **test d'intégration** vérifie que **plusieurs composants réels** fonctionnent ensemble : service + base de données, module + système de fichiers, deux services, code + bibliothèque externe.

| Les unitaires voient | Les intégrations voient |
|---|---|
| la logique de chaque pièce | les contrats entre pièces (ordre des arguments, formats, noms de colonnes) |
| rapide, précis | requêtes SQL réelles, transactions, contraintes |
| faux collaborateurs possibles | configuration, sérialisation, encodages |

## Les pièges classiques d'intégration

- ordre ou nom des paramètres ;

Le concept

# Les pièges classiques d'intégration (suite)

- format de date ou d'encodage (UTF-8 !) ;
- contrainte d'unicité ou clé étrangère violée ;
- transaction jamais validée (`commit` oublié) ou jamais annulée ;
- requête SQL correcte… sur une autre base de données que celle de production ;
- fuseau horaire, arrondi, tri.

## Une base de test fiable

| Technique | Avantage | Limite |
|---|---|---|
| **SQLite en mémoire** (`:memory:`) | rapide, isolée | dialecte différent de la production |
| **Base réelle jetable** (conteneur) | fidèle à la production | plus lent à démarrer |
| **Transaction annulée en fin de test** | très rapide | ne teste pas le `commit` réel |

Le concept

# Une base de test fiable (suite)

| Technique | Avantage | Limite |
|---|---|---|
| **Base recréée à chaque test** | isolation totale | un peu plus long |

Règle d'or : **chaque test crée ses données et ne dépend d'aucun autre**.

## Données de test

- **Fixtures** : jeu de données minimal construit par le test (ou une fonction fabrique).
- **Fabriques** (*builders*) : `un_membre(nom="Ada")` crée un objet valide avec des valeurs par défaut, et on ne précise que ce qui compte pour le test.
- **Jamais de données de production** réelles (confidentialité, mission 5 de HEF-100).

Le concept

# Fichiers et temps

- Fichiers : `tempfile.TemporaryDirectory()` (nettoyage automatique).
- Horloge : injectée (mission 9).
- Réseau : localhost, port éphémère (mission 5).

## Où placer la frontière ?

Un principe pratique : **la logique métier** se teste en unitaire (rapide, nombreux) ; **les adaptateurs** (base, fichiers, API) se testent en intégration (moins nombreux, plus réalistes). Un *test d'intégration étroit* vérifie un adaptateur avec sa vraie dépendance ; un *test d'intégration large* fait traverser plusieurs couches.

Le concept

# Où placer la frontière ? (suite)

🧠 ***Astuce*** *: écris un test d'intégration pour chaque* ***dépendance externe*** *que tu utilises (base, disque, API) : c'est là que se cachent les surprises.*

# Exemple pas à pas

**Problème** : tester un `DepotInscriptions` (SQLite) et un `ServiceInscription` ensemble, avec une base neuve par test, des contraintes réelles et un scénario complet.

```
import io, sqlite3, unittest

class DepotInscriptions:
    def __init__(self, connexion):
        self.bd = connexion
        self.bd.executescript('''
            CREATE TABLE IF NOT EXISTS cours (code TEXT PRIMARY KEY, titre TEXT NO
   T NULL, places INTEGER NOT NULL CHECK (places >= 0));
            CREATE TABLE IF NOT EXISTS inscriptions (
                membre TEXT NOT NULL, cours TEXT NOT NULL REFERENCES cours(code),
                PRIMARY KEY (membre, cours));
        ''')

    def ajouter_cours(self, code, titre, places):
        self.bd.execute("INSERT INTO cours VALUES (?, ?, ?)", (code, titre, places
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
    ))

    def places_restantes(self, code):
        ligne = self.bd.execute(
            "SELECT places - (SELECT COUNT(*) FROM inscriptions WHERE cours = ?) F
    ROM cours WHERE code = ?", (code, code)).fetchone()
        if ligne is None:
            raise KeyError(code)
        return ligne[0]

    def inscrire(self, membre, cours):
        self.bd.execute("INSERT INTO inscriptions VALUES (?, ?)", (membre, cours))

    def inscrits(self, cours):
        return [r[0] for r in self.bd.execute("SELECT membre FROM inscriptions WHE
    RE cours = ? ORDER BY membre", (cours,))]

class ServiceInscription:
    def __init__(self, depot):
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
        self.depot = depot

    def s_inscrire(self, membre, cours):
        if self.depot.places_restantes(cours) <= 0:
            raise RuntimeError("cours complet")
        try:
            self.depot.inscrire(membre, cours)
        except sqlite3.IntegrityError:
            raise RuntimeError("déjà inscrit") from None
        self.depot.bd.commit()

class TestInscriptionIntegration(unittest.TestCase):
    def setUp(self):
        self.bd = sqlite3.connect(":memory:")                    # base NEUVE pour
    chaque test
        self.bd.execute("PRAGMA foreign_keys = ON")
        self.depot = DepotInscriptions(self.bd)
        self.service = ServiceInscription(self.depot)
        self.depot.ajouter_cours("HEF-010", "Algorithmes", 2)
```

# Exemple pas à pas (suite)

```
    def tearDown(self):
        self.bd.close()

    def test_une_inscription_reduit_les_places(self):
        self.service.s_inscrire("ada", "HEF-010")
        self.assertEqual(self.depot.places_restantes("HEF-010"), 1)
        self.assertEqual(self.depot.inscrits("HEF-010"), ["ada"])

    def test_cours_complet_refuse_une_inscription_de_plus(self):
        self.service.s_inscrire("ada", "HEF-010")
        self.service.s_inscrire("bob", "HEF-010")
        with self.assertRaisesRegex(RuntimeError, "complet"):
            self.service.s_inscrire("cleo", "HEF-010")
        self.assertEqual(self.depot.inscrits("HEF-010"), ["ada", "bob"])

    def test_double_inscription_refusee_par_la_contrainte_de_la_base(self):
        self.service.s_inscrire("ada", "HEF-010")
        with self.assertRaisesRegex(RuntimeError, "déjà inscrit"):
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
            self.service.s_inscrire("ada", "HEF-010")

    def test_cours_inconnu_leve_keyerror(self):
        with self.assertRaises(KeyError):
            self.service.s_inscrire("ada", "NOPE")

    def test_la_base_refuse_une_inscription_a_un_cours_inexistant(self):
        with self.assertRaises(sqlite3.IntegrityError):
            self.depot.inscrire("ada", "NOPE")                  # clé étrangère :
    vue seulement avec la vraie base

    def test_le_commit_rend_l_inscription_visible_depuis_une_autre_connexion(self)
    :
        import tempfile, os
        with tempfile.TemporaryDirectory() as d:
            chemin = os.path.join(d, "test.db")
            c1 = sqlite3.connect(chemin)
            service = ServiceInscription(DepotInscriptions(c1))
            service.depot.ajouter_cours("HEF-020", "Structures", 5)
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
            c1.commit()
            service.s_inscrire("ada", "HEF-020")
            c2 = sqlite3.connect(chemin)                          # autre connexio
   n = autre « client »
            self.assertEqual(DepotInscriptions(c2).inscrits("HEF-020"), ["ada"])
            c1.close(); c2.close()

resultat = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.defaultTestL
   oader.loadTestsFromTestCase(TestInscriptionIntegration))
print("tests d'intégration :", resultat.testsRun, "| réussite :", resultat.wasSucc
   essful())
assert resultat.wasSuccessful(), resultat.failures + resultat.errors
```

**Lecture** :

- Chaque test crée une **base neuve** : aucune dépendance entre tests.

# Exemple pas à pas (suite)

- Certains comportements n'existent **que** grâce à la vraie base : la contrainte d'unicité qui produit « déjà inscrit », la clé étrangère. Un faux dépôt en mémoire ne les aurait pas vus.
- Le dernier test vérifie le `commit` avec **deux connexions** : un oubli de `commit` serait invisible avec une seule.

# À toi, avec aide

Ajoute à l'exemple une fonctionnalité **désinscription** et teste-la en intégration :

1. `depot.desinscrire(membre, cours)` ; si le membre n'est pas inscrit, lever `KeyError` ;
2. `service.se_desinscrire(membre, cours)` avec `commit` ;
3. tests : la désinscription libère une place ; désinscrire un absent lève `KeyError` ; après désinscription, une personne qui était refusée (cours complet) peut s'inscrire.

À toi, avec aide · réponse

# Indices progressifs

## Indice 1

`cursor.rowcount` indique combien de lignes ont été supprimées.

## Indice 2

Pense au `commit` dans le service.

## Indice 3

Crée une base neuve dans `setUp` comme dans l'exemple.

À toi, avec aide · réponse

# Réponse

```
import io, sqlite3, unittest

class Depot:
    def __init__(self, bd):
        self.bd = bd
        bd.executescript('''CREATE TABLE cours (code TEXT PRIMARY KEY, places INTE
   GER);
                            CREATE TABLE inscriptions (membre TEXT, cours TEXT, PR
   IMARY KEY (membre, cours));''')
    def places_restantes(self, code):
        return self.bd.execute("SELECT places - (SELECT COUNT(*) FROM inscriptions
    WHERE cours=?) FROM cours WHERE code=?", (code, code)).fetchone()[0]
    def inscrire(self, m, c):
        self.bd.execute("INSERT INTO inscriptions VALUES (?, ?)", (m, c))
    def desinscrire(self, m, c):
        if self.bd.execute("DELETE FROM inscriptions WHERE membre=? AND cours=?",
   (m, c)).rowcount == 0:
            raise KeyError((m, c))
```

# Réponse (suite)

```
class Service:
    def __init__(self, depot): self.depot = depot
    def s_inscrire(self, m, c):
        if self.depot.places_restantes(c) <= 0:
            raise RuntimeError("complet")
        self.depot.inscrire(m, c); self.depot.bd.commit()
    def se_desinscrire(self, m, c):
        self.depot.desinscrire(m, c); self.depot.bd.commit()

class TestDesinscription(unittest.TestCase):
    def setUp(self):
        self.bd = sqlite3.connect(":memory:")
        self.depot = Depot(self.bd)
        self.service = Service(self.depot)
        self.bd.execute("INSERT INTO cours VALUES ('C1', 1)")
    def tearDown(self): self.bd.close()

    def test_la_desinscription_libere_une_place(self):
        self.service.s_inscrire("ada", "C1")
```

À toi, avec aide · réponse

# Réponse (suite)

```
        self.service.se_desinscrire("ada", "C1")
        self.assertEqual(self.depot.places_restantes("C1"), 1)

    def test_desinscrire_un_absent_leve_keyerror(self):
        with self.assertRaises(KeyError):
            self.service.se_desinscrire("ada", "C1")

    def test_un_refuse_peut_s_inscrire_apres_une_desinscription(self):
        self.service.s_inscrire("ada", "C1")
        with self.assertRaises(RuntimeError):
            self.service.s_inscrire("bob", "C1")
        self.service.se_desinscrire("ada", "C1")
        self.service.s_inscrire("bob", "C1")
        self.assertEqual(self.depot.places_restantes("C1"), 0)

r = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.defaultTestLoader.l
   oadTestsFromTestCase(TestDesinscription))
assert r.wasSuccessful(), r.failures + r.errors
print("3 tests d'intégration verts")
```

À toi, avec aide · réponse

# Réponse (suite)

Le troisième test est un **scénario** : il enchaîne plusieurs opérations réelles, ce qu'un test unitaire d'une seule méthode ne fait pas.

# À toi, en solo

**Test d'intégration de fichiers.** Écris `ArchiveurJournaux` qui lit un dossier de journaux `.log` (une ligne par événement `NIVEAU message`) et produit un fichier `resume.json` avec le nombre d'événements par niveau et la première ligne d'erreur. Teste avec de **vrais fichiers** dans un dossier temporaire :

1. plusieurs fichiers, dont un vide et un avec des lignes mal formées (comptées à part) ;
2. un fichier d'un autre encodage (UTF-8 avec accents) ;
3. un dossier inexistant (erreur claire) ;
4. exécution deux fois : le résultat doit être identique (idempotence) ;
5. le fichier `resume.json` est écrit **atomiquement** (écriture dans un fichier temporaire puis renommage) : teste qu'aucun fichier temporaire ne reste.

À toi, en solo · réponse

# Pistes de correction

```
import json, os, tempfile, unittest, io
from collections import Counter

class ArchiveurJournaux:
    def __init__(self, dossier):
        self.dossier = dossier

    def resumer(self):
        if not os.path.isdir(self.dossier):
            raise FileNotFoundError(f"dossier introuvable : {self.dossier}")
        niveaux, mal_formees, premiere_erreur = Counter(), 0, None
        for nom in sorted(os.listdir(self.dossier)):
            if not nom.endswith(".log"):
                continue
            with open(os.path.join(self.dossier, nom), encoding="utf-8") as f:
                for ligne in f:
                    ligne = ligne.rstrip("\n")
                    if not ligne:
                        continue
```

# Pistes de correction (suite)

```
                    niveau, _, message = ligne.partition(" ")
                    if niveau not in ("INFO", "WARN", "ERROR") or not message:
                        mal_formees += 1
                        continue
                    niveaux[niveau] += 1
                    if niveau == "ERROR" and premiere_erreur is None:
                        premiere_erreur = message
        resume = {"niveaux": dict(sorted(niveaux.items())), "mal_formees": mal_for
    mees, "premiere_erreur": premiere_erreur}
        temporaire = os.path.join(self.dossier, "resume.json.tmp")
        with open(temporaire, "w", encoding="utf-8") as f:
            json.dump(resume, f, ensure_ascii=False, sort_keys=True)
        os.replace(temporaire, os.path.join(self.dossier, "resume.json"))
        return resume

class TestArchiveur(unittest.TestCase):
    def setUp(self):
        self.tmp = tempfile.TemporaryDirectory()
        self.d = self.tmp.name
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
    def tearDown(self): self.tmp.cleanup()
    def ecrire(self, nom, texte):
        with open(os.path.join(self.d, nom), "w", encoding="utf-8") as f: f.write(
texte)

    def test_comptage_par_niveau_et_lignes_mal_formees(self):
        self.ecrire("a.log", "INFO démarrage\nERROR échec connexion à la base\nWAR
N lent\n")
        self.ecrire("vide.log", "")
        self.ecrire("b.log", "n'importe quoi\nINFO ok\n")
        r = ArchiveurJournaux(self.d).resumer()
        self.assertEqual(r["niveaux"], {"ERROR": 1, "INFO": 2, "WARN": 1})
        self.assertEqual(r["mal_formees"], 1)
        self.assertEqual(r["premiere_erreur"], "échec connexion à la base")

    def test_accents_conserves_dans_le_resume(self):
        self.ecrire("a.log", "ERROR café brûlé\n")
        ArchiveurJournaux(self.d).resumer()
        with open(os.path.join(self.d, "resume.json"), encoding="utf-8") as f:
```

# Pistes de correction (suite)

```
            self.assertIn("café brûlé", f.read())

    def test_dossier_inexistant(self):
        with self.assertRaises(FileNotFoundError):
            ArchiveurJournaux(os.path.join(self.d, "nope")).resumer()

    def test_idempotent_et_sans_fichier_temporaire(self):
        self.ecrire("a.log", "INFO x\nERROR y\n")
        a = ArchiveurJournaux(self.d)
        self.assertEqual(a.resumer(), a.resumer())
        self.assertNotIn("resume.json.tmp", os.listdir(self.d))

r = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.defaultTestLoader.l
   oadTestsFromTestCase(TestArchiveur))
assert r.wasSuccessful(), r.failures + r.errors
print("4 tests d'intégration sur de vrais fichiers")
```

À toi, en solo · réponse

# Pistes de correction (suite)

Ces tests utilisent de **vrais fichiers** dans un dossier jetable : ils capturent l'encodage, le renommage atomique et les fichiers oubliés, qu'un faux système de fichiers masquerait.

# Pièges classiques

| Piège | Pourquoi c'est un problème | Comment l'éviter |
|---|---|---|
| **Tester l'intégration avec des faux partout** | Les vraies coutures ne sont pas testées | Au moins un test avec les vraies dépendances |
| **Base partagée entre tests** | Ordre et pollution des données | Base neuve ou transaction annulée par test |
| **Tout en intégration** | Suite lente, diagnostic flou | Pyramide : logique en unitaire, adaptateurs en intégration |
| **Données de production dans les tests** | Fuite de données personnelles | Données fabriquées, minimales |

Mini-boss

# Mini-boss

**Intégration complète de HashBox.** Construis trois composants : `DepotMembres` (SQLite), `DepotCours` (SQLite) et `ServiceInscription` (règles : capacité, un seul cours simultané au même horaire, prérequis).

1. Écris le schéma avec contraintes (clés étrangères, unicité, CHECK) ;
2. écris des **fabriques** (`un_membre()`, `un_cours()`) ;
3. écris 12+ tests d'intégration, dont 3 scénarios de bout en bout ;
4. écris un test qui **détecte un oubli de `commit`** ;
5. mesure le temps de la suite (objectif < 1 s) et explique comment tu la garderais rapide à 500 tests.

Mini-boss · réponse

# Pistes de correction

Un bon travail : base en mémoire neuve dans `setUp`, `PRAGMA foreign_keys = ON`, fabriques avec valeurs par défaut valides et paramètres nommés, tests des contraintes (clé étrangère, unicité, CHECK de capacité), scénarios (inscription → complet → désinscription → promotion), test à deux connexions sur un fichier temporaire pour le `commit`, chronométrage avec `time.perf_counter()`. Pour rester rapide : base en mémoire, schéma créé une fois puis copié ou transaction annulée, tests indépendants parallélisables, peu de tests larges.

Quiz éclair

# Quiz éclair

**1.** Un test d'intégration vérifie surtout :

A. que des composants réels fonctionnent ensemble
B. la vitesse du processeur
C. l'orthographe des messages

**2.** Pour isoler les tests de base de données, on peut :

A. partager une seule base entre tous les tests
B. créer une base neuve (ou annuler la transaction) pour chaque test
C. tester en production

**3.** Un test avec un faux dépôt ne détecterait pas :

A. une faute de frappe dans le nom d'un test

Quiz éclair

# Quiz éclair (suite)

B. une ligne vide
C. une contrainte d'unicité de la vraie base

**4.** 🔁 *Révision mission 03* : F.I.R.S.T. désigne les qualités d'un test :

A. rapide, indépendant, reproductible, auto-vérifiant, écrit à temps
B. fort, intuitif, riche, simple, testable
C. fiable, interne, résilient, sûr, tracé

**5.** 🔁 *Révision mission 03* : Une couverture de 100 % signifie :

A. que le code n'a aucun bug
B. que chaque ligne a été exécutée, pas forcément bien vérifiée
C. que les tests sont rapides

Quiz éclair · réponse

# Réponses

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

# Je réfléchis

1. Quelle dépendance externe de ton projet (base, fichier, API) n'est jamais testée avec la réalité ?
2. Quand un faux collaborateur t'a-t-il déjà donné une fausse confiance ?

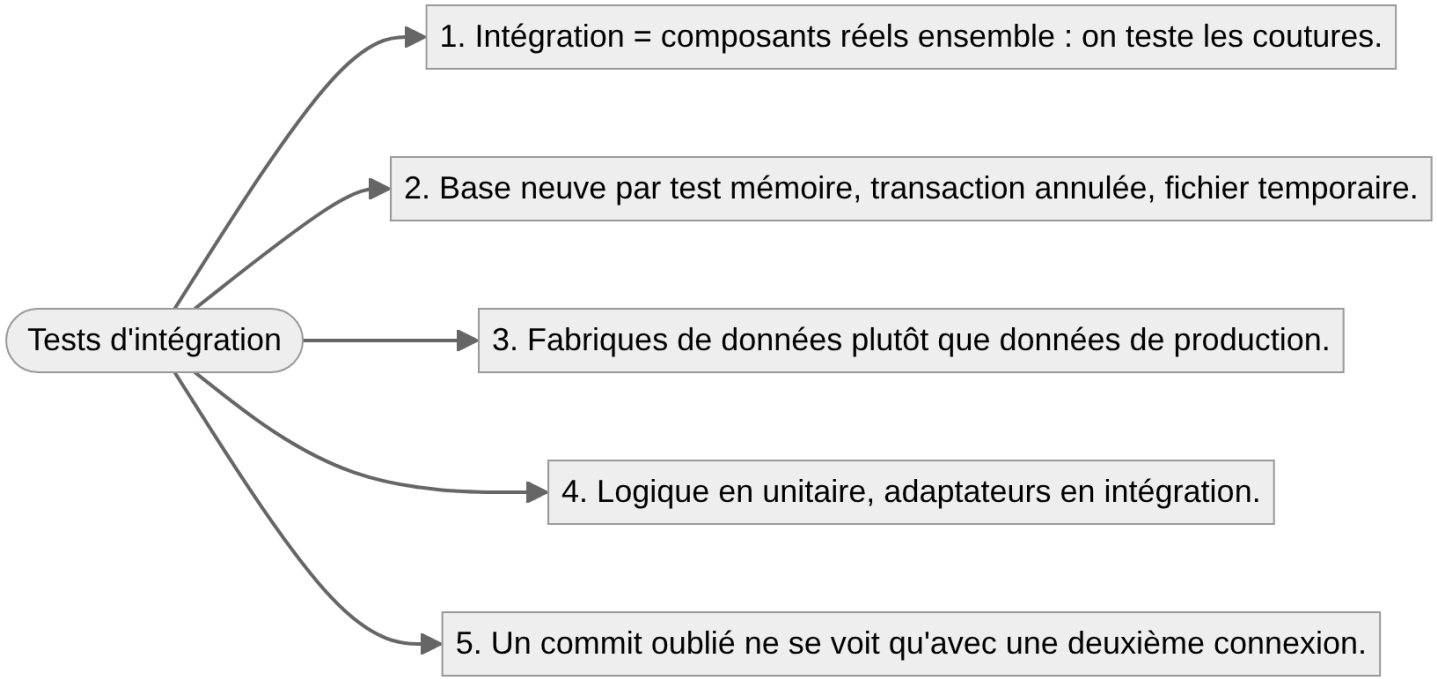
À retenir

# À retenir

1. Intégration = composants réels ensemble : on teste les coutures.
2. Base neuve par test (mémoire, transaction annulée, fichier temporaire).
3. Fabriques de données plutôt que données de production.
4. Logique en unitaire, adaptateurs en intégration.
5. Un `commit` oublié ne se voit qu'avec une deuxième connexion.

À retenir

# Le schéma de la mission



MISSION ACCOMPLIE

# +100 XP

## Badge débloqué : Assembleur de preuves

Prochaine mission : Tests API et contrats

