# TP4 - Tests d'intégration (TypeScript)

## Objectifs

- Ecrire des tests d'integration pour un **repository** avec un store en memoire et SQLite
- Tester une **API REST** avec Supertest et Express
- Decouvrir le **Property-Based Testing** avec fast-check
- Decouvrir le **Contract Testing** avec Pact

## Mise en place

Clonez le projet de demarrage :

```
git clone https://lab.frogg.it/Edouard_Mangel/tests-validation-tp4-ts.git
cd tests-validation-tp4-ts
npm install
npm run build
```

Le projet contient :

- `src/Todo.ts` — l'entite
- `src/TodoRepository.ts` — le repository avec les operations CRUD (version memoire et SQLite)
- `src/TodoService.ts` — les regles metier (titre obligatoire, detection de retard)
- `src/app.ts` — l'application Express avec les endpoints REST
- `src/PricingService.ts` — un service de calcul de prix (pour l'exercice 3)

Creez le projet de test :

```
npm install -D vitest supertest @types/supertest better-sqlite3 @types/better-sql
npm install -D fast-check @pact-foundation/pact
```

Ajoutez dans `package.json` :

```
{
  "scripts": {
```

```
    "test": "vitest",
    "test:run": "vitest run"
  }
}
```

## Exercice 1 - Tests du repository (35 min)

### Objectif

Tester `TodoRepository` avec un store en memoire, puis avec SQLite in-memory. Comparer les resultats.

### Contraintes

- Utiliser `beforeEach` ou `afterEach` pour l'isolation entre les tests
- Chaque test doit etre isole (pas d'etat partage entre les tests)
- Tester au minimum : `add`, `getById`, `getAll`, `update`, `delete`

### Verification

- [ ] Tous les tests passent avec le store en memoire
- [ ] Au moins un test se comporte differemment avec SQLite (contraintes, types)
- [ ] Vous pouvez expliquer : "qu'est-ce que le store en memoire ne detecte pas que SQLite detecte ?"

### Indices

▶ Indice 1 — Isolation avec le store en memoire
▶ Indice 2 — SQLite in-memory avec better-sqlite3
▶ Indice 3 — Difference de comportement

---

## Exercice 2 - Tests d'API avec Supertest (35 min)

### Objectif

Tester les endpoints REST de l'application Express via Supertest.

### Contraintes

- Creer une factory qui instancie l'app avec un repository en memoire
- Tester les scenarios suivants :
  - `GET /api/todos` → liste vide (200)
  - `POST /api/todos` puis `GET /api/todos` → le todo cree est dans la liste
  - `DELETE /api/todos/:id` puis `GET /api/todos/:id` → 404
  - `GET /api/todos/999` → 404
- Verifier les **codes HTTP** ET le **contenu** de la reponse

### Verification

- [ ] Les tests passent et verifient les codes de statut
- [ ] Vous avez decouvert le bug de serialisation de dates
- [ ] Vous pouvez repondre : "qu'est-ce que ce test a attrape que le test unitaire n'aurait pas vu ?"

### Indices

▶ Indice 1 — Supertest avec Express

▶ Indice 2 — Envoyer et lire du JSON

▶ Indice 3 — Le bug de serialisation

---

## Exercice 3 - Property-Based Testing avec fast-check (25 min)

### Objectif

Ecrire des proprietes fast-check pour `PricingService`.

Le `PricingService` fourni dans le projet a deux methodes :

- `calculateTotal(prices: number[], discountPercent: number)` — calcule le total avec remise
- `serializeTodo(todo: Todo)` / `deserializeTodo(json: string)` — serialisation JSON

```
npm install -D fast-check
```

### Contraintes

- Ecrire au moins **2 proprietes** :
  - Une propriete de type **roundtrip** (serialize puis deserialize redonne l'objet original)
  - Une propriete de type **invariant** (le prix est toujours >= 0, quel que soit le discount)
- Utiliser `fc.assert(fc.property(...))` au lieu d'un test par l'exemple
- Chaque propriete doit generer au moins 100 cas (comportement par defaut de fast-check)

### Verification

- [ ] fast-check genere >= 100 cas par propriete
- [ ] En cas d'echec, le shrinking montre le cas minimal
- [ ] Vous comprenez la difference entre un test par l'exemple et un test de propriete

### Indices

▶ Indice 1 — Syntaxe de base
▶ Indice 2 — Propriete roundtrip
▶ Indice 3 — Gerer les edge cases

---

## Exercice 4 - Contract Testing avec Pact (15 min)

### Objectif

Ecrire un test consommateur Pact pour un service externe `InventoryService`.

### Contexte

Le `TodoApi` consomme un service `InventoryService` pour verifier la disponibilite de ressources. Le `InventoryClient` est deja present dans le projet :

```
// src/InventoryClient.ts
export interface InventoryItem {
    id: number;
    name: string;
    quantity: number;
    isAvailable: boolean;
}
```

```
export class InventoryClient {
    constructor(private baseUrl: string) {}

    async getItem(id: number): Promise<InventoryItem | null> {
        const response = await fetch(`${this.baseUrl}/inventory/${id}`);
        if (!response.ok) return null;
        return response.json();
    }
}
```

## Guidage

Cet exercice est plus guide car le concept est nouveau.

**1.** Installez PactNet :

```
npm install -D @pact-foundation/pact
```

**2.** Completez le squelette suivant dans `tests/ConsumerPactTests.test.ts` :

```
import { PactV4, MatchersV3 } from '@pact-foundation/pact';
import { InventoryClient } from '../src/InventoryClient';
import path from 'path';

const provider = new PactV4({
    consumer: 'TodoApi',
    provider: 'InventoryService',
    dir: path.resolve(process.cwd(), 'pacts'),
});

describe('InventoryClient - contrat Pact', () => {
    it("getItem - article existant - retourne l'article", async () => {
        await provider
            .addInteraction()
            .given("l'article 1 existe")
            .uponReceiving("une requete pour l'article 1")
            .withRequest('GET', '/inventory/1')
            .willRespondWith(200, {}, MatchersV3.like({
                id: 1,
                name: 'Stylo',
                quantity: 42,
                isAvailable: true,
            }))
            .executeTest(async (mockServer) => {
                const client = new InventoryClient(mockServer.url);
```

```
            const item = await client.getItem(1);

            expect(item).not.toBeNull();
            expect(item?.name).toBe('Stylo');
            expect(item?.quantity).toBe(42);
        });
    });

    // A vous : ecrivez un deuxieme test pour le cas ou l'article n'existe pas (4
});
```

**3.** Lancez le test. Observez le fichier `.pact.json` genere dans le dossier `pacts/`.

**4.** Reflexion : que se passerait-il si l'equipe de `InventoryService` renommait le champ `quantity` en `stock` ? Comment le contract testing detecterait-il ce probleme ?

---

## Reflexion de fin (10 min)

Remplissez ce tableau en vous basant sur votre experience pendant ce TP :

| Bug / Probleme | Decouvert par quel niveau de test ? | Pourquoi ce niveau ? |
|---|---|---|
| Requete qui retourne `undefined` au lieu de tableau vide | | |
| Perte de timezone dans la serialisation JSON | | |
| Violation de contrainte NOT NULL | | |
| Changement de format d'une API externe | | |
| Calcul de prix negatif avec un discount > 100% | | |
| Le service n'appelle pas la bonne methode du repository | | |

---

## Recapitulatif

| Concept C# (.NET) | Equivalent TypeScript | Ou l'avez-vous pratique ? |
|---|---|---|
| EF Core InMemoryDatabase | Store en memoire (tableau `Todo[]`) | Exercice 1 |
| EF Core SQLite in-memory | better-sqlite3 `new Database(':memory:')` | Exercice 1 |
| `IClassFixture<T>` avec base de donnees | `beforeEach` / `afterEach` | Exercice 1 |
| InMemory vs SQLite (limites) | Meme constat | Exercice 1 |
| `WebApplicationFactory<Program>` | `createApp(repo)` + Supertest | Exercice 2 |
| `CustomWebApplicationFactory` | Injection du repo en memoire dans `createApp` | Exercice 2 |
| Tests HTTP (codes + contenu) | `request(app).get(...).expect(200)` | Exercice 2 |
| Bug de serialisation | Meme constat (timezone JSON) | Exercice 2 |
| `[Property]` avec FsCheck | `fc.assert(fc.property(...))` fast-check | Exercice 3 |
| Propriete roundtrip | Meme concept | Exercice 3 |
| Propriete invariant | Meme concept | Exercice 3 |
| Shrinking | Meme concept (fast-check le fait automatiquement) | Exercice 3 |

| | | |
|---|---|---|
| Contract Testing avec PactNet | `@pact-foundation/pact` | Exercice 4 |
| Fichier `.pact` | Fichier `.pact.json` dans `pacts/` | Exercice 4 |