TP4 - Tests d'integration

Objectifs

- Ecrire des tests d'integration pour un **repository** avec EF Core InMemory et SQLite
- Tester une **API REST** avec `WebApplicationFactory`
- Decouvrir le **Property-Based Testing** avec FsCheck
- Decouvrir le **Contract Testing** avec Pact

Mise en place

Ouvrez le projet `TodoApi` fourni dans le dossier `tp-starter/`.

```
cd tp-starter/TodoApi
dotnet restore
dotnet build
```

Le projet contient :

- `Todo.cs` — l'entite
- `AppDbContext.cs` — le contexte EF Core
- `TodoRepository.cs` — le repository avec les operations CRUD
- `TodoService.cs` — les regles metier (titre obligatoire, detection de retard)
- `TodosController.cs` — les endpoints REST
- `PricingService.cs` — un service de calcul de prix (pour l'exercice 3)

Creez le projet de test :

```
dotnet new xunit -n TodoApi.Tests
dotnet add TodoApi.Tests reference TodoApi
dotnet add TodoApi.Tests package Microsoft.EntityFrameworkCore.InMemory
```

```
dotnet add TodoApi.Tests package Microsoft.EntityFrameworkCore.Sqlite
dotnet add TodoApi.Tests package Microsoft.AspNetCore.Mvc.Testing
```

Exercice 1 - Tests du repository (35 min)

Objectif

Tester `TodoRepository` avec EF Core InMemory, puis avec SQLite in-memory. Comparer les resultats.

Contraintes

- Utiliser `IClassFixture<T>` ou `IDisposable` pour le setup de la base
- 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 InMemory
- [ ] Au moins un test se comporte differemment avec SQLite (contraintes, types)
- [ ] Vous pouvez expliquer : "qu'est-ce que InMemory ne detecte pas que SQLite detecte ?"

Indices

▶ Indice 1 — Isolation avec InMemory
▶ Indice 2 — SQLite in-memory
▶ Indice 3 — Difference de comportement

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

Objectif

Tester les endpoints REST de `TodosController` via `WebApplicationFactory<Program>`.

Contraintes

- Creer un `CustomWebApplicationFactory` qui remplace la base par InMemory
- Tester les scenarios suivants :
  - `GET /api/todos` → liste vide (200)
  - `POST /api/todos` puis `GET /api/todos` → la todo creee 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 — CustomWebApplicationFactory

▶ Indice 2 — Envoyer et lire du JSON

▶ Indice 3 — Le bug de serialisation

Exercice 3 - Property-Based Testing avec FsCheck (25 min)

Objectif

Ecrire des proprietes FsCheck pour `PricingService`.

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

- `CalculateTotal(List<decimal> prices, decimal discountPercent)` — calcule le total avec remise
- `SerializeTodo(Todo todo)` / `DeserializeTodo(string json)` — serialisation JSON

```
dotnet add TodoApi.Tests package FsCheck.Xunit
```

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 `[Property]` au lieu de `[Fact]`
- Chaque propriete doit generer au moins 100 cas

Verification

- [ ] FsCheck 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 :

```
public class InventoryClient
{
    private readonly HttpClient _client;

    public InventoryClient(HttpClient client)
    {
        _client = client;
    }

    public async Task<InventoryItem?> GetItem(int id)
    {
        var response = await _client.GetAsync($"/inventory/{id}");
        if (!response.IsSuccessStatusCode) return null;
        return await response.Content.ReadFromJsonAsync<InventoryItem>();
    }
}

public class InventoryItem
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public int Quantity { get; set; }
    public bool IsAvailable => Quantity > 0;
}
```

Guidage

Cet exercice est plus guide car le concept est nouveau.

1. Installez PactNet :

```
dotnet add TodoApi.Tests package PactNet
```

2. Completez le squelette suivant dans `ConsumerPactTests.cs` :

```
using PactNet;

public class InventoryConsumerTests
{
    private readonly IPactBuilderV4 _pactBuilder;

    public InventoryConsumerTests()
    {
        var pact = Pact.V4("TodoApi", "InventoryService", new PactConfig()
        _pactBuilder = pact.WithHttpInteractions();
    }

    [Fact]
    public async Task GetItem_ExistingItem_ReturnsItem()
    {
        // 1. Definir l'interaction attendue
        _pactBuilder
            .UponReceiving("a request for an existing inventory item")
            .WithRequest(HttpMethod.Get, "/inventory/1")
            .WillRespond()
            .WithStatus(System.Net.HttpStatusCode.OK)
            .WithJsonBody(new
            {
                id = 1,
                name = "Stylo",
                quantity = 42
            });

        // 2. Executer le test avec le mock server Pact
        await _pactBuilder.VerifyAsync(async ctx =>
        {
            var client = new HttpClient { BaseAddress = ctx.MockServerUri
            var inventoryClient = new InventoryClient(client);

            var item = await inventoryClient.GetItem(1);

            Assert.NotNull(item);
            Assert.Equal("Stylo", item.Name);
            Assert.Equal(42, item.Quantity);
        });
    }
```

```
    // A vous : ecrivez un deuxieme test pour le cas ou l'item n'existe pa
}
```

3. Lancez le test. Observez le fichier `.pact` 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 LINQ qui retourne null au lieu de 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	Ou l'avez-vous pratique ?
EF Core InMemoryDatabase	Exercice 1
SQLite in-memory	Exercice 1
`IClassFixture<T>` avec base de donnees	Exercice 1
InMemory vs SQLite (limites)	Exercice 1
`WebApplicationFactory<Program>`	Exercice 2
`CustomWebApplicationFactory`	Exercice 2
Tests HTTP (codes + contenu)	Exercice 2
Bug de serialisation	Exercice 2
`[Property]` avec FsCheck	Exercice 3
Propriete roundtrip	Exercice 3
Propriete invariant	Exercice 3
Shrinking	Exercice 3
Contract Testing avec Pact	Exercice 4
Fichier `.pact`	Exercice 4