R4.02 – TP 5

TESTS AVEC DBSETUP ET ASSERTJ-DB

Table des matières

Objectif

L'objectif est de réaliser des tests avec DbSetup et avec AssertJ-DB.

A lire avec attention

Ne pas oublier de démarrer Docker.

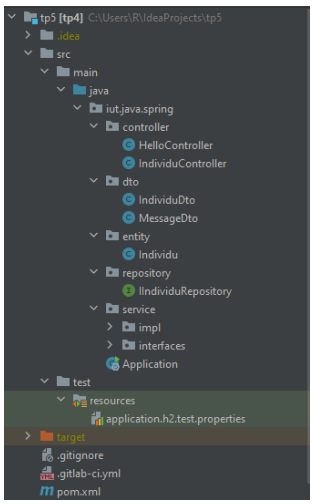
Les énoncés sont à lire avec attention : le respect ou non-respect des indications sera pris en compte dans la notation

Les principales informations pour réaliser le TP sont dans le support de cours. Il n'est pas exclu de compléter/confirmer certaines de ces informations grâce à Internet.

Travaux Pratiques

Création du dépôt et ajout des sources

1. Créer un dépôt « `TP5` » dans le groupe précédemment créé et le cloner.
2. Ajouter les sources contenues dans « `TP5.zip` » et les pousser sur GitLab



Ajout de la chaine CI/CD

1. Ajouter une chaîne d'intégration continue pour exécuter les tests dans le GitLab Runner sous Docker avec Maven.
2. Commiter et vérifier l'exécution de la chaine CI/CD.
3. Si création de la chaine CI/CD depuis GitLab, récupérer les sources depuis GitLab pour ne pas avoir de conflits.

Ajout d'un jeu de données aléatoires individuelles

1. Reprendre les données générées dans la TP 4

Test d'intégration de méthode de service listant les individus

Indice(s) : Pages 65 à 67 du cours

1. Créer une classe « `IndividuServiceTest` » dans le package « `iut.java.spring.tests.integration` » du dossier « `src/test/java` »
2. Dans cette classe, charger le jeu de données avec DbSetup
3. Ajouter l'injection grâce à Spring d'un attribut d'instance de type « `IIndividuService` »

4. Créer une méthode de test (**avec AssertJ**) nommée « `testGetList` » sur la méthode « `getList` » de « `IIndividuService` »
5. Le test doit vérifier le nom de l'ensemble des individus retournés par « `getList` »
6. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
7. Cette méthode devra être commentée (pas besoin de faire un roman)
8. Commiter et vérifier l'exécution de la chaine CI/CD

Test fonctionnel de lecture d'un individu et de l'état de la table

Indice(s) : Page 68 à 77 du cours. Hibernate initialise la base de données en créant les tables avec les colonnes par ordre alphabétique (sauf l'identifiant qui est premier). Le constructeur Table(Source source, String name, String[] columnsToCheck, String[] columnsToExclude) permet de choisir l'ordre des colonnes, isEqualTo(DateValue expected)

1. Créer une classe « `IndividuControllerTableTest` » dans le package « `iut.java.spring.tests.functional` » du dossier « `src/test/java` »
2. Dans cette classe, charger le jeu de données avec DbSetup
3. Créer une méthode de test (**avec AssertJ**) nommée « `testGet` » qui effectue un appel HTTP GET vers la méthode « `get` ».
4. Le test doit vérifier toutes les valeurs de l'individu retourné
5. Le test doit vérifier l'état de la base de données après le test (s'assurer que rien n'a été modifié) en utilisant Table.
6. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
7. Cette méthode devra être commentée (pas besoin de faire un roman)
8. Commiter et vérifier l'exécution de la chaine CI/CD

Test fonctionnel de suppression d'un individu et de l'état de la table

1. Toujours dans cette classe, créer une méthode de test (**avec AssertJ**) nommée « `testRemove` » qui effectue un appel HTTP DELETE vers la méthode « `remove` ».
2. Le test doit vérifier l'état de la base de données après le test (s'assurer que l'enregistrement prévu a été supprimé) en utilisant Table.
3. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
4. Cette méthode devra être commentée (pas besoin de faire un roman)

Test fonctionnel de création d'un individu et de l'état d'une requête

1. Créer une classe « `IndividuControllerRequestTest` » dans le package « `iut.java.spring.tests.functional` » du dossier « `src/test/java` »
2. Dans cette classe, charger le jeu de données avec DbSetup
3. Créer une méthode de test (**avec AssertJ**) nommée « `testAdd` » qui effectue un appel HTTP POST vers la méthode « `add` ».
4. Le test doit vérifier toutes les valeurs (à l'exception de l'Id) de l'individu retourné
5. Le test doit vérifier l'état de la base de données après le test (s'assurer que l'enregistrement prévu a été créé) en utilisant Request.
6. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
7. Cette méthode devra être commentée (pas besoin de faire un roman)

Test fonctionnel de modification d'un individu trouvé et de l'état d'une requête

1. Toujours dans cette classe, créer une méthode de test (**avec AssertJ**) nommée « `testModifyFound` » qui effectue un appel HTTP PUT vers la méthode « `modify` » dans le cas où l'identifiant de l'individu envoyé existe dans la base de données.

2. Le test doit vérifier l'état de la base de données après le test (s'assurer que l'enregistrement prévu a été modifié) en utilisant Request.
3. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
4. Cette méthode devra être commentée (pas besoin de faire un roman)

Test fonctionnel de modification d'un individu non trouvé et de l'état d'une requête

1. Toujours dans cette classe, créer une méthode de test (**avec AssertJ**) nommée « `testModifyNotFound` » qui effectue un appel HTTP PUT vers la méthode « `modify` » dans le cas où l'identifiant de l'individu envoyé n'existe pas dans la base de données.
2. Le test doit vérifier l'état de la base de données après le test (s'assurer que rien n'a été modifié) en utilisant Request.
3. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
4. Cette méthode devra être commentée (pas besoin de faire un roman)

Test fonctionnel de suppression d'un individu et des changements

1. Créer une classe « `IndividuControllerChangesTest` » dans le package « `iut.java.spring.tests.functional` » du dossier « `src/test/java` »
2. Dans cette classe, charger le jeu de données avec DbSetup
3. Créer une méthode de test (**avec AssertJ**) nommée « `testRemove` » qui effectue un appel HTTP DELETE vers la méthode « `remove` ».
4. Le test doit vérifier les changements durant le test (s'assurer que l'enregistrement prévu a été supprimé) en utilisant Changes.
5. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
6. Cette méthode devra être commentée (pas besoin de faire un roman)

Test fonctionnel de création d'un individu et des changements

1. Toujours dans cette classe, créer une méthode de test (**avec AssertJ**) nommée « `testAdd` » qui effectue un appel HTTP POST vers la méthode « `add` ».
2. Le test doit vérifier toutes les valeurs (à l'exception de l'Id) de l'individu retourné
3. Le test doit vérifier les changements durant le test (s'assurer que l'enregistrement prévu a été créé) en utilisant Changes.
4. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
5. Cette méthode devra être commentée (pas besoin de faire un roman)

Test fonctionnel de modification d'un individu trouvé et des changements

1. Toujours dans cette classe, créer une méthode de test (**avec AssertJ**) nommée « `testModifyFound` » qui effectue un appel HTTP PUT vers la méthode « `modify` » dans le cas où l'identifiant de l'individu envoyé existe dans la base de données.
2. Le test doit vérifier les changements durant le test (s'assurer que l'enregistrement prévu a été modifié) en utilisant Changes.
3. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
4. Cette méthode devra être commentée (pas besoin de faire un roman)

Test fonctionnel de modification d'un individu non trouvé et des changements

1. Toujours dans cette classe, créer une méthode de test (**avec AssertJ**) nommée « `testModifyNotFound` » qui effectue un appel HTTP PUT vers la méthode « `modify` » dans le cas où l'identifiant de l'individu envoyé n'existe pas dans la base de données.
2. Le test doit vérifier les changements durant le test (s'assurer que rien n'a été modifié) en utilisant Changes.

3. Cette méthode devra découper les parties « ARRANGE », « ACT » et « ASSERT ».
4. Cette méthode devra être commentée (pas besoin de faire un roman)