SENG275 – Lab 9
Due Wednesday March 31, 11:55 pm

Technical note: You now have another repository on gitlab.csc.uvic.ca, named database_testing.git. This repository has a lot of dependencies, so the first time you clone and build it, it might take a while to download and prepare all the modules.

Welcome to Lab 9. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm. This lab is worth 4% of your final grade.

Quick summary of what you need to do:

1. Write the integration/database tests provided in Exercise 1. You'll need to modify both src/test/…/DatabaseTest.java and src/main/java/…/data/StudentRepository.java. Commit and push your changes back as usual.

You're done!

Integration and Database Testing

Imagine your team has been working for a while testing various parts of a student-management program for a University. Some testers have been working on the web UI, others have been testing the course registration rules, etc. Each module of this large piece of software has been tested in isolation so far, but now it's time to begin combining them into larger and larger sub-programs. Managing this process of bringing together these different modules is called *integration*.

Our task for this lab is to address one issue the team has identified. We would like to test the interface between the program, and the database which will be persistantly storing all our student information. The team doesn't yet want to incorporate the actual, disk-based database into the testing – doing so would be *system* testing, and the program isn't ready for that yet.

Instead, the team would like to temporarily replace the full, disk-based database with a lighter, in-memory one. Ideally, you'll write tests for this temporary database, confirm that it works, and then when it's time for a system test, you can swap the full database back in and the tests will continue to pass – if they do, then the main program and the database will have been successfully integrated.

This process may remind you of Mocking – in integration testing, we'll be substituing an entire portion of our application (the database) rather than a few classes. Also, when we write a mock, we specify what data we get back from the doubled object. In this case, our test double is an entire, fully-functional third-party database; hopefully equal in capability to our permanent repository, just lighter to invoke and easier to configure.

In this lab we'll be using Spring Boot (a lightweight version of the extremely popular Spring java application framework), and h2, an in-memory database.

Advantages

- A lightweight, in-memory database like h2database is ready to accept transactions almost instantly. A large, disk-based database may take some time before it's ready for interaction.
- A full-featured, sophisticated database may require expert configuration, and may have thousands of options that need to be considered. A lightweight database can usually do a pretty good job on default settings.
- If our tests crash, a remote or independent database instance may remain running, especially if it's located on another machine. We can be certain that our in-memory database has released its memory when our test completes.

Disadvantages

- Most obviously, you are testing against a different database than the one you intend to deploy. Each database has different features, and your tests may inadvertantly use some capability that isn't available on your deployment database. Worse, no relational database fully implements the SQL standard – all implement some subset of SQL, and almost all have vendor-specific extensions to the standard.
- Tests may pass BECAUSE of the in-memory nature of the lightweight substitute. For example, if your test is supposed to fail if a query takes too long, it may pass BECAUSE memory is faster than the disk or network – it may be impossible to test these sorts of physical dependencies until working with the production database.

Advantages and disadvantages like these make excellent midterm questions.

Student.java

Each student has an id number, a firstName, a lastName, and an activeStatus (either 1 or 0). We have public getters for these properties, but no public setters, so we cannot change a student's information once created. This means that we cannot Update a Student (Update is one of the four capabilities we look for in a proper persistent storage scheme, along with Create, Read and Delete). The annotation for Id tells us that the unique Id number for each student will be automatically generated – AUTO means they'll be generated in ascending numerical order).

StudentRepository.java

What's going on with StudentRepository? It's an interface, so there's no code for the methods, and we can't create a StudentRepository directly – one will be created for us by Spring using the @Autowired annotation in our testing file. In the interface declaration, we're saying that we'd like a database that contains Students, and its primary key (its unique identifier for each entry) is a Long (which is Student.Id). Spring then goes ahead and (at runtime) generates a class for us and does the work of providing an instance for us that provides the actual functionality that we specify in outline in the StudentRepository interface.

Note the SQL Query specified by the annotation for findAllActiveStudents(). This SQL query will be run against the database every time this method is called – the process of generating the list of active students will be run as SQL, not Java code. We should expect a simple query like this to work with nearly every SQL database, but more complicated ones may fail, as we mentioned.

Exercise 1:

Write the following tests in DatabaseTest.java. The various annotations at the top of the file create the database and provide a variable of type StudentRepository for you – all your access to the database will be done through that variable. (4%)

1. Write a test called createTest() to test the studentRepository.save() functionality. Add one student to the database using this method and confirm that the student is there and that their name and active status are correct.
2. Write a test called deleteTest() to test the studentRepository.delete() functionality. Perform the actions and assertions you believe are necessary and sufficient to confirm that the database can indeed delete.
3. Write a test called findByLastNameTest() to test the findByLastName() functionality. Add at least six students with different last names to the database in this test, then confirm that you can find one by last name.
4. Write a test called findAllActiveStudentsTest() to test the method of the same name from StudentRepository.java. You should add at least six students to the database – three of them should be active, three inactive. Test that this method gives you the expected list of students.
5. Write a test called sortedFirstNamesTest(). This test should put at least six students into the database, each with a different first name. The method findAllFirstNameSort() should return all the students in order, sorted by their first name. This exercise will require you to write a @Query annotation in StudentRepository.java, similar to the one written for findAllActiveStudents(). Hint: SQL uses the 'order by' keywords to sort records by some column. Experiment and see what you can do.

This concludes the lab content for SENG275. On behalf of the teaching team, thank you for your efforts this semester, and for your participation in these labs. Best of luck and stay safe!