**Conventions/Configurations to follow before testing:**

* All the tests should be run against and empty database so as the existing data which might be wrong then you will get the wrong output.
* Update the test-db-context.xml as well whenever you change anything in db-context.xml you can just replace the content just make sure you are not overriding anyone else changes.
* Do the same thing with the mappers in test\resources.
* Update the schema.sql with the updated table definitions whenever you have any update on any table or create any new one.
* Create the stubs to add values to all the tables in Dummy data generator you will be making use of that in the tests.
* Check the existing createUser which returns a new user after creating it.
* Always extend the BaseTest java class when you are creating the new test class
* Don’t change any of the configurations and base class without prior notice or discussion might mess up or break anything.

**Unit testing workflow:**

**Services(Assuming you want to test UserService):**

* Create the test class with the same name with the Test appended at the beginning eg. TestUserService must extend BaseTest
* Auto wire the Service you want to test like below in this case UserService as we want to test that one.

**@Autowired(required = true)**

**private UserService userService;**

* **Testing Insert**: For testing the insert first get the User from the dummy data generator it is already defined in the base test class you can get its reference by using ddg.
  + use the function from userService to save the user to test db
  + use assertNotNull and assertTrue methods from Junit api to see whether the number of rows affected is null
  + Cross check the database insert by fetching all users from db (using findAllUsers function) and following the mentioned assert methods as well.
  + finally remove the dummy data inserted to db using deleteUser function in userService

* **Testing Get**: For testing the get we first insert the data and then query the db to fetch the data and match the data that is inserted and fetched if they both match test passes else test fails.
* **Testing Update**: For testing the insert first get the User from the dummy data generator it is already defined in the base test class you can get its reference by using ddg.

**Commonly used methods and usage:**

* assertSame("Objects must be the same", expected, actual);
* assertEquals("Objects must be equal", expected, actual);
* assertNull("Object must be null", actual);
* assertNotNull("Object must not be null", actual);
* assertTrue("Objects must be the same", expected == actual);
* assertTrue("Objects must be equal", expected.equals(actual));
* assertTrue("Object must be null", actual == null);
* assertTrue("Object must not be null", actual != null);
* **fail(String)** - Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have a failing test before the test code is implemented. The String parameter is optional.
* **assertTrue([message], boolean condition)** - Checks that the boolean condition is true.
* **assertFalse([message], boolean condition)** - Checks that the boolean condition is false.
* **assertEquals([String message], expected, actual)** - Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
* **assertEquals([String message], expected, actual, tolerance)** - Test that float or double values match. The tolerance is the number of decimals which must be the same.
* **assertNull([message], object)** - Checks that the object is null.
* **assertNotNull([message], object)** - Checks that the object is not null.
* **assertSame([String], expected, actual)** - Checks that both variables refer to the same object.
* **assertNotSame([String], expected, actual)** - Checks that both variables refer to different objects.