Section: Technical sciences

Korsun Anna
Student of the
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine

Shtets Kateryna
Student of the
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine

BACKEND TESTING RULES AND AUTOMATED QUALITY ASSURANCE APPROACHES

Each product requires high-quality support. Due to the 3 depth levels diagram of the testing: unit, integration, and system testing, it is known, that the system-verification tests are more expensive and less successful. This follows that lower-level tests are more accurate and convenient for maintaining product quality. Manual testing, in general, provides UI whole system verification, including white-box and black-box approaches. To reduce manual testing and initialize a lower level of quality assurance of whole back-end and front-end parts were decided to set up the automation test project.

Initially, it is needed to create Quality Assurance documents based on which automated test cases would be created. Each test case must describe isolated functional way how the user would interact with particular part of program. The good test case consists of steps to reproduce and expected result [1].

Start up project does not require long term quality support. That is why automated quality assurance engineer must be consentrated on creating smoke testing flow, avoiding edge cases coverage.

Remember that the goal of automated test cases is not only to cover functional cases but also to combine test design with verification of all page objects, APIs, database connections, calculation methods, etc. Refering to 3 depth levels diagram, most part of automated test must be dedicated to database and API, less one for page elements.

Basical example with Login form, where email and password fields are presented. First test case, is positive – try to login with valid user creadentials. Second one is negative – login by non registered previously email. Last one is attempt to sign in with valid email and wrong password. Note, with such test case set we cover all general user business needs; all components: users` related API, database, page objects would be affected [2].

The project is based on 3 level architecture: API test level; Business layer - to process methods which are related to both database and API, general calculations, etc; and connect to the database layer. To organize appropriate work with MS SQL DB were decided to use the Dapper simple object mapping tool. It is designed primarily to be used in scenarios where you want to work with data in a strongly-typed fashion - as business objects in a .NET application. NUnit framework is used to specify test`s body and successfuly execute it as separate part of project. To optimize code usage and reduce time of each test execution session, next NUnit attributes tools are used:

– [Parallelizable] attribute is set before each test-suite, after whole project executing, separate test cases from each test-suite are started on own stream; such optimization reduces execution time and test data overlaying;

– [OneTimeSetUp] attribute also used once per test suite in separate method, where precondition actions for each test are described;

– [TearDown] attribute for method, where actions after each test case is finished are described: chrome driver session closing, deleting test date, taking screenshots, etc.

Interaction with page objects are provided by using Selenium and Chrome Driver. After UI test is loaded, selenium session creates: test actions are executed on imitation of chrome browser. Built-in Selenium Framework element functions imitate user actions.

Also, during software development, developers also need to test their code. Most often this is done using unit and integration tests. Let's start with unit testing. There are a few rules that a developer should keep in mind when writing unit tests.

The principles of writing quality tests were not invented from scratch. Most experienced developers know about them in the same way that they know about design principles, but as with design principles, it was Bob Martin who combined the five principles of testing, resulting in the resounding name F.I.R.S.T. (Stand for: Fast, Independent, Repeatable, Self-Validating, Timely) [3].

So, each unit test should have the following characteristics:

– fast. Tests must run quickly;

– independent. The results of one test should not be input to another;

– repeatable. Tests should produce the same results regardless of the runtime environment. The results should not depend on whether they are running on your local machine or on the build server;

– self-validating. The result of the test must be a boolean value. A test either passes or fails and should be easy to understand for any developer. You don't have to force people to read logs just to determine if a test passed or failed;

– timely. Tests must be created in a timely manner. The untimeliness of writing tests is the main reason that they are postponed until later, and this "later" never comes. Even if you do not write tests before the code, they should be written at least in parallel with the code.

After we have run and validated the unit tests for our web API, we can consider another type of testing [4]. We used unit testing to test and validate expectations for the solution's internals. When we are satisfied with the quality of

the internal tests, we can move on to testing the API from the front end, which is called integration testing.

Integration tests should be written and run after all components are completed so that your API can be used with the correct HTTP response to test. Unit tests should be viewed as testing independent and isolated code segments, while integration tests are used to test the entire logic of each API on our HTTP endpoint.

Literature

1. Graham D., Black R., Van Veenendaal E. Foundations of software testing. – Cengage, 2019.
2. Axelrod A. Complete Guide to Test Automation //Matan: Apress. 2018.
3. Martin R. C. The clean coder: a code of conduct for professional programmers. Pearson Education, 2011.
4. Osherove R. The Art of Unit Testing: with examples in C. Simon and Schuster, 2013.