The different types of software testing

Compare different types of software testing, such as unit testing, integration testing, functional testing, acceptance testing, and more!

BY STEN PITTET

There are many different types of testing that you can use to make sure that changes to your code are working as expected. Not all testing is equal, though, and we will see here how the main testing practices differ from each other.

Manual vs. automated testing

At a high level, we need to make the distinction between manual and automated tests. Manual testing is done in person, by clicking through the application or interacting with the software and APIs with the appropriate tooling. This is very expensive as it requires someone to set up an environment and execute the tests themselves, and it can be prone to human error as the tester might make typos or omit steps in the test script.

Automated tests, on the other hand, are performed by a machine that executes a test script that has been written in advance. These tests can vary a lot in complexity, from checking a single method in a class to making sure that performing a sequence of complex actions in the UI leads to the same results. It's much more robust and reliable than automated tests – but the quality of your automated tests depends on how well your test scripts have been written.

Automated testing is a key component of continuous integration and continuous delivery and it's a great way to scale your QA process as you add new features to your application. But there's still value in doing some manual testing with what is called exploratory testing as we will see in this guide.

The different types of tests

Unit tests

Unit tests are very low level, close to the source of your application. They consist in testing individual methods and functions of the classes, components or modules used by your software. Unit tests are in general quite cheap to automate and can be run very quickly by a continuous integration server.

Integration tests

Integration tests verify that different modules or services used by your application work well together. For example, it can be testing the interaction with the database or making sure that microservices work together as expected. These types of tests are more expensive to run as they require multiple parts of the application to be up and running.

Functional tests

Functional tests focus on the business requirements of an application. They only verify the output of an action and do not check the intermediate states of the system when performing that action.

There is sometimes a confusion between integration tests and functional tests as they both require multiple components to interact with each other. The difference is that an integration test may simply verify that you can query the database while a functional test would expect to get a specific value from the database as defined by the product requirements.

End-to-end tests

End-to-end testing replicates a user behavior with the software in a complete application environment. It verifies

that various user flows work as expected and can be as simple as loading a web page or logging in or much more complex scenarios verifying email notifications, online payments, etc...

End-to-end tests are very useful, but they're expensive to perform and can be hard to maintain when they're automated. It is recommended to have a few key end-to-end tests and rely more on lower level types of testing (unit and integration tests) to be able to quickly identify breaking changes.

Acceptance testing

Acceptance tests are formal tests executed to verify if a system satisfies its business requirements. They require the entire application to be up and running and focus on replicating user behaviors. But they can also go further and measure the performance of the system and reject changes if certain goals are not met.

Performance testing

Performance tests check the behaviors of the system when it is under significant load. These tests are non-functional and can have the various form to understand the reliability, stability, and availability of the platform. For instance, it can be observing response times when executing a high number of requests, or seeing how the system behaves with a significant of data.

Performance tests are by their nature quite costly to implement and run, but they can help you understand if new changes are going to degrade your system.

Smoke testing

Smoke tests are basic tests that check basic functionality of the application. They are meant to be quick to execute, and their goal is to give you the assurance that the major features of your system are working as expected.

Smoke tests can be useful right after a new build is made to decide whether or not you can run more expensive tests, or right after a deployment to make sure that they application is running properly in the newly deployed environment.

How to automate your tests

An individual can execute all the tests mentioned above, but it will be very expensive and counter-productive to do so. As humans, we have limited capacity to perform a large number of actions in a repeatable and reliable way. But a machine can easily do that rapidly and will test that login/password combination works for the 100th time without complaining.

To automate your tests, you will first need to write them programmatically using a testing framework that suits your application. PHPUnit, Mocha, RSpec are examples of testing frameworks that you can use for PHP, Javascript, and Ruby respectively. There are many options out there for each language so you might have to do some research and ask developer communities to find out what would be the best framework for you.

When your tests can be executed via script from your terminal, you can have them be automatically executed by a continuous integration server like Bamboo or use a cloud service like Bitbucket Pipelines. These tools will monitor your repositories and execute your test suite whenever new changes are pushed to the main repository.

If you're just getting started with testing, you can read our continuous integration tutorial to help you with your first

continuous integration tutorial to help you with your first test suite.

Exploratory testing

The more features and improvements go into your code, the more you'll need to test to make sure that all your system works properly. And then for each bug you fix, it would be wise to check that they don't get back in newer releases. Automation is key to make this possible and writing tests sooner or later will become part of your development workflow.

So the question is whether it is still worth doing manual testing? The short answer is yes, and it should be focused on what is called exploratory testing where the goal is to uncover non-obvious errors.

An exploratory testing session should not exceed two hours and need to have a clear scope to help testers focus on a specific area of the software. Once all testers have been briefed, is up to them to try various actions to check how the system behaves. This type of testing is expensive by nature but is quite helpful to uncover UI issues or verify complex user workflows. It's something especially worth doing whenever a significant new capability is added to your application to help understand how it behaves under edge cases.

A note about testing

To finish this guide, it's important to talk about the goal of testing. While it's important to test that users can use your application (I can log in, I can save an object) it is equally important to test that your system doesn't break when bad data or unexpected actions are performed. You need to anticipate what would happen when a user makes a typo, tries to save an incomplete form or uses the wrong API. You need to check if someone can easily compromise data, get access to a resource they're not supposed to. A good testing suite should try to break your app and help understand its limit.

And finally, tests are code too! So don't forget them during code review as they might be the final gate to production.

STEN PITTET

I've been in the software business for 10 years now in various roles from development to product management. After spending the last 5 years in Atlassian working on Developer Tools I now write about building software. Outside of work I'm sharpening my fathering skills with a wonderful toddler.