ToDoApp Automated Testing

Automation testing simplifies the testing effort to its minimal, The Idea is to cut short the QA round so the application or release can be delivered in minimal turnaround time.

Increase of test coverage

Sufficient test coverage of software projects is often achieved only with great effort. Frequent repetition of the same or similar test cases is laborious and time consuming to perform manually. Some examples are:

- Regression test after debugging or further development of software
- Testing of software on different platforms or with different configurations
- Data-driven testing (creation of tests using the same actions but with many different inputs)

Test automation allows performing different types of testing efficiently and effectively.

Here are the best practice of test automation

- Test automation require technical knowledge of coding and system.
- Start with manual testing and analyze the manual test cases which can be automated.
- Most important part of automation is automation strategy. Automation need to be very structured. Identify what to test and how.
- Don’t automate those part of UI, that is enhancing in every release.
- If you are planning to automate any part of application, make sure you have **build automation** and **continuous integration** in-place.

This document describes different types of automated tests that can be performed for ToDoApp which includes functional and non-functional tests.

Functional Testing

Validating an application or Web site conforms to its specifications and correctly performs all its required functions. This entails a series of tests which perform a feature by feature validation of behavior, using a wide range of normal and erroneous input data. This can involve **testing of the product's user interface, APIs, database management, security, installation, networking**, etc

Functional testing can be performed on an automated or manual basis using black box or white box methodologies.

Non-Functional Testing

In most testing methodologies, functional testing involves testing the application against the business requirements. Functional testing is done using the functional specifications provided by the client or by using the design specifications like use cases provided by the design team.

On the other hand, non-functional testing involves testing the application against the non-functional requirements, which typically involve measuring/testing the application against defined technical qualities. Some examples of non-functional testing are described below:

- **Performance, Load, Stress Testing**
- **Security, Vulnerability Testing**
- **Usability Testing**
- **Compatibility Testing**

Automated Testing

Types of Functional and Non-Functional Automated Testing that can be performed for toDoApp:

There are at least 4 types of automated tests: **unit, integration, UI, and performance**. **In general, these 4 of types tests are a very natural path to follow for automation**:

- **Unit --> Integration:** It's necessary to have a simple unit test before complex integration tests
- **Integration --> UI:** Integrate the backend (APIs or web services, database) before UI tests.
- **UI --> Performance:** Functionally run the code from end-to-end and perform UI tests before expecting reliable performance measures on it.

The higher you go, the tests are more expensive: Unit tests (low-level) are cheapest, performance tests (high-level) are most expensive. So it's bad business to pay for an integration test to do the work of a unit test.

Unit Testing:

A unit test is a script that exercises a specific code object by initializing it, calling methods or functions, and checking the return values. Programmers write and run these tests locally as part of a **test driven development process**.

Automated Unit Tests are quick to develop and can be easily incorporated into Continuous Integration process. **These tests should comprise the largest portion of automated tests.** I would encourage software developers to create lots of Unit Tests and to perform test-driven development with all new software code.

Of all the types of automated testing, this is the one that should get the most use because developers should be running these tests at least daily, if not more frequently. Developers should also be writing new tests for all new code that they write and all changes that they make. It has been our experience that this also is an area of testing that many teams overlook or do not adequately implement. An organization might have experience with integration or performance testing, but unit testing often falls by the wayside.

Unit testing is **good for Units of in-memory code, like parsing, calculations, validations, algorithms, formatting**, etc.
Example: Field validation: verify every input combination for a text field.

Because unit tests are in-memory, they usually run very fast, and hence can be run upon check-in and with each build. Therefore they provide the "**first level of defense**" to ensure code continues to functionally work.

As an example, let's consider a simple class with a method that adds an object to the end of a list. Without unit testing, you probably would have performed some basic testing to make sure that the object was added to the end of the list, not to the beginning or somewhere in the middle. Unit testing provides the confidence that the method will do the proper thing, no matter what the input is. For example, consider what the sample method would do if the inputs were as follows:

- The object is null.
- The list is null.
- The list is empty.
- The list has exactly one item.

These are the types of tests that often are not conducted. At some point, a section of code that calls the sample class might be modified so that the list can be empty. The application no longer operates correctly, but the bug is more difficult to locate because what appeared to be working code that has not been changed is now broken. Rightly so, the focus is placed on the new code that was written rather than the code where the bug actually exists. By ensuring that the code can properly deal with all types of inputs, this method can be used in a variety of circumstances.

Unit Testing Tool: JUnit / TestNG

Although many techniques can be used to implement unit testing, the most popular tool for Java development is Junit/TestNG. This tool provides a framework for developing unit tests, which fits very nicely into the Maven build and deployment process.

Integration/ Smoke Testing:

While Unit testing deals with testing cases in isolation, integration testing or API testing combines individual software modules and tests as a group. The purpose is to ensure that a set

of functions and classes can work together, with data being successfully passed from one to the other. Automated Integration Tests need to be performed for all areas that cannot be Unit Tested.

These tests are much more difficult to write, run more slowly, and require a lot of coding. Given the tremendous resources required, it's critical to prioritize which Integration Tests to automate using a prioritized automation backlog.

An integration test is a code level script (i.e., one that doesn't run the normal UI) that tests a complete process involving multiple objects ensuring high-level flows work, such as you can call a web service that loads or saves data to a database and writes something to a file..

An integration test requires more work to build because you have to set up more data, and you might need a mockup database. However, there are frameworks like **Cucumber** that walk you through the process.

Key Features of Cucumber Framework Used

1. Gherkin used for easily readable test steps based on BDD approach.
2. Test cases and test data are stored in feature file.
3. Test script code saved in step definition file.
4. Page object model to make the code reusable.
5. Focus on popular open source tools: Jenkins, Selenium, Cucumber.
6. Test result reports in HTML format with screenshots of failed steps for debugging.

Integration tests have a medium level of efficiency, and you should build them for important actions if your programming framework supports them.

Service integration tests are important for managing large projects. Service integration tests run the complete app, including calls to multiple Web services. They often run on a centralized test system that has its own substantial database.

The following are popular **automated integration testing tools**:

- Jbehave
- Cucumber

Smoke Test: A Smoke Test is a high-level, preliminary test to reveal simple failures severe enough to reject a prospective software release. Create an automated Smoke Test to add to the Continuous Integration process.

To determine other Integration Tests to automate, start an automation backlog. Then, prioritize your backlog items. First, assess the monetary value and focus on the highest return items. Next, look at high usage areas of your software and then determine what browsers need to be

tested. Note the top couple of browsers, but focus on the top browser for now. This is the browser you will start using for automation.

Lastly, review your bug list for risky areas of the software that tend to break frequently. Compare this to your backlog. If an issue has happened numerous times and has the potential to occur again, move that backlog item up in priority. If issues keep cropping up that are related to a specific browser, move that browser up in the backlog as well. Once an automated Integration Test is stable, add this automation to your Continuous Integration process.

Automated Web Service / API Tests

Application Programmable Interfaces (API's) are collections of procedures & functions that can be used by other applications to fulfill their functionality. It is a collection of methods wrapped in a library that interact with the application to meet a functional requirement. Each API is supposed to behave the way it is coded, i.e. it is functionality specific. These APIs may offer different results for different type of the input provided. The errors or the exceptions returned may also vary. However once integrated within a product, the common functionality covers a very minimal code path of the API and the functionality testing / integration testing may cover only those paths. By considering each API as a black box, a generalized approach of testing can be applied. But, there may be some paths which are not tested and lead to bugs in the application. Applications can be viewed and treated as APIs from a testing perspective.

What we test here is usually the backend functionality, compliance and security issues.

In web applications, we can test the Request and Response of our application that whether they are secure and encrypted or not. The most popular tool for API testing is SOAPUI which has both free and paid versions.

We may test our REST API using tools like postman or other tools. It is really fine to manually testing our API endpoints, but how to test a hundred or maybe thousand endpoints and keep it reliable? or imagine every endpoint has at least 2 or 3 scenarios and we have to cover all of scenarios every single time.

In order to test REST APIs, **REST Assured library** can be used. REST-assured provides a lot of nice features, such as **DSL-like syntax**, **XPath-Validation**, **Specification Reuse**, **easy file uploads** and with those features we will handle automated API testing much easier.
Rest Assured has **a gherkin type syntax** which is used in **BDD** (**B**ehavior **D**riven **D**evelopment).

The RESTful API test automation can be integrated with the client's continuous integration tool, Jenkins, in order to become part of their smoke testing on daily builds.

UI Automation / Regression Testing

The User Interface (UI) also needs to be automated in order to exercise all the functionality and paths of the application. UI automation is tricky because it's usually comprised of numerous dependencies. For example, some functionality of the system may only be accomplished by following a complex sequence of UI events. Since the GUI may change significantly across versions of the application, UI tests can frequently break, so this portion of automation should make up the smallest percentage of your automated tests.

UI testing is also closest to what users will do with our application. Since the user will use the mouse and keyboard, automated UI tests also mimic the same behaviour by making use of mouse and keyboard to click or write to objects present on the user interface. Due to this, we can find bugs early and it can be used in many scenarios such as regression testing or filling up forms which takes too much time.

UI / Functional tests are the slowest of all the tests because functional tests are generally standing up the real application, either on the desktop or in a browser, and driving that user interface around to perform the same actions a user would.

An example of this would be using something like **Selenium** to start up a browser, open your web application, and take some actions to create a task or subtask in the ToDoApp.

.

A great many tools and frameworks can be used to help you write functional tests depending on what platform you're working on. There are many flavors of functional tests, but at their core they're responsible for helping you verify one specific test case around your system's functionality behaves as expected.

Regression Testing

Similar in scope to a functional test, a regression test allows a consistent, repeatable validation of each new release of a product or Web site. Such testing ensures reported product defects have been corrected for each new release and that no new quality problems were introduced in the maintenance process. Though regression testing can be performed manually an automated test suite is often used to reduce the time and resources needed to perform the required testing.

A full regression test runs through all of the application functionality to make sure that you did not break something that seems unrelated to your changes. Full regression tests are not very

efficient, and they take a long time to run. If you want to release frequently or continuously, you should not require a full regression test. If you think you need a full regression tests to prevent unexpected bugs, you should add more test layers until you do not need the full regression test.

Load/Performance Testing

Load testing is a generic term covering Performance Testing and Stress Testing.

Performance Testing

Performance testing can be applied to understand your application or web site's scalability, or to benchmark the performance in an environment of third party products such as servers and middleware for potential purchase.

This sort of testing is particularly useful to identify performance bottlenecks in high use applications. Performance testing generally involves an automated test suite as this allows easy simulation of a variety of normal, peak, and exceptional load conditions.

These are long running tests and need to be run multiple times a day within the acceptable time frame of a critical production fix.

Stress Testing

Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails and how. A graceful degradation under load leading to non-catastrophic failure is the desired result. Often Stress Testing is performed using the same process as Performance Testing but employing a very high level of simulated load.

The '**Apache JMeter**' application, is an open source software, a 100% pure 'Java application' designed to load test functional behavior and measure performance.

Security Testing

Software applications are getting complex and can potentially get threatened due to market risks and various inherent vulnerabilities. Testing, thus, has to be rigorous and iterative
Automated security tests can be run in regressions in protections around cross site scripting, SQL injection, and any of the other numerous (and really scary) security-related threats.

Automation for security tests is similar to automation of functional or performance tests. While automating the tests, security tests can be segmented into functional Security tests such as authentication and password generation, Specific non-functional tests against known weaknesses, Security scanning of the application and infrastructure, and Security testing application logic.

From an automation point of view, security tests can be categorised as follows:

1. Functional Security Tests.

These are essentially the same as automated acceptance tests, but targeted at verifying that security features such as authentication and logout, work as expected. They can mostly be automated using existing acceptance testing browser automation tools like **Selenium/WebDriver.**

2. Specific non-functional tests against known weaknesses.

- Includes testing known weaknesses and mis-configurations such as lack of the HttpOnly flag on session cookies, or use of known weak SSL suites and ciphers. These are particularly well suited for automation because the weaknesses are known up front (if not by the development team, then by the security team). What's more is that these tests lend themselves to a TDD approach in that they can **serve as the security specification** before building the application and environment.
- Some work has already been done in extracting these types of tests into security test automation frameworks, see: BDD-Security, Mittn by F-Secure and Gauntlt.
- Because these test non-functional aspects of the application, they need access to the HTTP layer which browser automation tools do not provide. So testing these requires a hybrid approach: Browser automation together with a proxy server to inspect and inject requests

3. Security scanning of the application and infrastructure.

- Burp Intruder (Commercial) and OWASP ZAP (Open Source) are application scanners which inspect and test at the HTTP layer by injecting attack data into parameters and evaluating the application's response. They **can** provide in-depth security scanning if they're used correctly. But if they're simply used to spider the application and run an automated test then there's a good chance that they won't find or test all the available content.
- To successfully automate application scanning, one should ensure that all of the content to be scanned is navigated and populated in the scanning tool, before starting to spider and scan the application. If you already have acceptance tests that drive a browser, then these can be re-used to populate the Burp or ZAP content before kicking off a scan.

4. Security testing application logic.

Automated tools can only go so far in detecting security flaws. To identify flaws in the logic of the application requires a human brain. These require ingenuity and experience to find, but once the attack is defined they too can be recorded as automated tests and become a part of the security regression tests.

Production logging and monitoring

All software should have an automated system for logging errors and reporting them automatically. You should also be collecting post-release user activity and system performance. You should be collecting user feedback and responding to it.

As you improve these capabilities and improve your reaction time, you will have a built-in system for eliminating defects and improving usability, performance and profitability. No matter what your pre-release testing plan, you should be trying to gather as much data as possible about the post-release performance of your software, and you should be trying to reduce the time that it takes your team to respond to this data.

This can be a very efficient type of testing. If your users can tolerate some errors, you can use it as a substitute for other tests. If your system is online, you can deploy to production before you merge to the mainline. Then, you watch the production system to make sure it works correctly. If there is a problem, you roll back to the mainline version. If it works correctly, you merge to mainline. This technique gives you a cleaner mainline that has code which has already been tested in production. It makes rollbacks simpler, and it gives the other developers a more stable mainline.
.