Test Approach for Stock Tracker App

TEST STRATEGY

The test strategy is the high-level description of the test requirements from which a detailed test plan can later be derived, specifying individual test scenarios and test cases. Our primary concern is functional testing – ensuring that the API and Web UI functions correctly.

- to ensure that the implementation is working correctly as expected — no bugs!
- to ensure that the implementation is working as specified according to the requirements specification
- to prevent regressions between code merges and releases.

Before any implementation test can begin, it is important to make sure that the contract is correct. That can be done first by inspecting the spec and making sure that endpoints are correctly named, that resources and their types correctly reflect the object model, that there is no missing functionality or duplicate functionality, and that relationships between resources are reflected in the API and Web - UI correctly.
If this is a customer-facing public API, this might be your last chance to ensure that all contract requirements are met, because once the API is published and in use, any changes you make might break customers' code.

What to test?

Now that we have validated the contract, we are ready to think of what to test. All the tests can be classified into various categories or buckets that can be scoped differently in terms of ownership. For example, unit testing and integration testing are primarily owned by developers. It is there responsibility to ensure they have complete unit testing as part of the task definition and code reviews. At a next stage is to consider the important nonfunctional requirements of the system, like performance and load tests. These are again good candidates for the development team to own the testing as part of their pipelines to ensure they continuously adhere to the SLA and any failures should block the pipeline. The challenge here is to ensure we have an infrastructure that can simulate and mimic the load and measure the results.
At the top of the test pyramid is the Functional end-to-end tests that covers all the end user test cases as a customer and which will cover the work flow between web UI and the underlying API's. So, as part of this test pyramid strategy, we classify our test categories as below.

1. Functional end-to-end tests
2. Web UI test
3. API tests
4. Performance / Latency measurements
5. Stress / Load tests
6. Integration testing
7. Unit testing

Factors to be considered - Test Automation

Key factors required for a Test Automation to be successful include:

- Committed Management
- Budgeted Cost
- Process

- Dedicated Resources
- Realistic Expectations

Realistic Expectations

Management and the project team must keep realistic expectations and should keep them in mind during the entire test automation life-cycle.

- Achieving 100% automatic tests is an unreachable goal
- All tests cannot be automated
- Benefits of automation is reaped only after several cycles of test execution
- No immediate payback for the investment
- Ramp-up time will be required for tool selection, framework creation
- Record and Playback helps minimally in test automation
- No available tool in the market supports all the systems and GUI objects
- Not all the testers can write scripts. Availability of specialized resources is a must.

Why - Framework

A framework defines the organization's way of doing things - a 'Single Standard'. Following this standard would result in the project team achieving:

Test Library - Process Definition

Standard Scripting and Team Consistency

Encapsulation from Complexities

Scripts and Data Separation

- Automation test scripts separated from input data store (for example: XML, Excel files)
- No modification is required to the test scripts
- Only input data gets manipulated for testing with multiple input values

Implement and Maximize Re-Usability

- Establish the developed libraries across the organization/project team/product team, i.e. publish the library and provide access rights
- Utilities/components shared across the team
- Usage of available libraries
- Minimized effort for repeated regression cycles

Extensibility and Maintenance

Design Test Automation Framework

Generally, testers start creating test scripts based on the scenarios. This includes multiple actions to be performed against each object. This approach leads to an ad-hoc test script creation and duplicate testing effort, i.e. testers, would create test scripts for a single action in different scenarios.

Our approach takes a different path as explained below. For designing a framework, various elements need to be taken into consideration. Utilities/Components (re-usable) would be designed for the following elements that include (not limited to):

- Actions to be performed - Identification of actions to be automated for each object of the application
- Communicating Systems - Study of different internal systems, third-party systems and their communication methodology
- Business Rules - List of business layers and any specific algorithm has to be studied. A separate function needs to be created for each specific algorithm.
- Database Communication - Database validation and check point validations
- Communication with additional automation tools - In the scenario, where we would require communicating with different automation tool. All the communication requirements needs to be identified and designed
- Data retrieval - Retrieval of data from multiple input data stores
- Schedulers - Functionalities related to invoking of relevant scripts based on scheduler configuration
- Tool Extensibility - Overcoming tool limitations. Components for actions/validations for which the tool does not provide any support
- Device Communication - Device communication and data transfer related actions/validations
- Log - User-defined logs for analysis
- Error Handlers - Error handlers to handle known and unknown errors and log the information
- Custom Messages - Display of relevant defined messages
- Result Presentation - Customized and presentable reports on completion of test execution

Test automation framework would be designed based on the listed factors, using the following guidelines.

- Application-independent.
- Easy to expand, maintain, and perpetuate.
- Encapsulate the testers from the complexities of the test framework
- Identify and abstract common functions used across multiple test scripts
- Decouple complex business function testing from navigation, limit-testing, and other simple verification and validation activities.
- Decouple test data from the test scripts
- Structure scripts with minimal dependencies - Ensuring scripts executing unattended even on failures

Summary: Achieve an Ideal Test Pyramid

We would never ship a feature without the proper metrics, monitors, alarms, runbooks, and other operational mechanisms, so too should we never be willing to ship a feature without automated testing. Teams should mechanize this in their SDLC by requiring automated quality work to be defined as early as possible in the process, and to track it at the lowest level possible. Quality should always be a part of every person's role, from requirements, to launch, to operations. Development teams are not well-positioned to implement tests higher in the test pyramid. This naturally tends them to depend on manual testing to gain confidence to go ahead launching a feature. The practice can become a bottleneck for feature launches. Ideally, we need to minimize end-to-end testing and tend towards automation that is fast, reliable and can isolate failures. Achieving a strong "test pyramid" is critical. We should stop using end-to-end tests, especially manual regression and end-to-end testing, as our primary feedback loop. We need more automated tests earlier in the pipeline. This test strategy tries to fill this gap by reducing the dependency on Manual testing and enabling testing at different levels to achieve automation with faster feedback loops at all levels of the test pyramid. And minimize the dependency on e2e testing.