**Software Testing**

*(From various sources)*

Content

1. Test Case, Test Scenario
2. Unit Testing, API Testing

**Test Case, Test Scenario**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | | | | |
| - Set of actions executed to verify a particular feature/functionality of software application  - Contains test steps, test data, precondition, postcondition  - Developed for specific test scenario to verify any requirement  - Includes specific variables/conditions which a testing engineer can compare expected and actual results to determine whether a software product is functioning as per requirements of customer  - Mostly derived from Test Scenarios  - Helps in exhaustive testing of an application | | | **Component** | **Example** |
| **Prerequisites** | *Access to Chrome Browser* |
| **Test Data** | *Userid = …* |
|  | *Password = …* |
| **Test Scenario** | *Verify on entering valid userid and password, the customer can login* |
| **Step #** | *1* |
| **Step Details** | *Navigate to http://...* |
| **Expected Results** | *Site should be open* |
| **Actual Results** | *As Expected* |
| **Pass / Fail / Not executed / Suspended** | *Pass* |
| **Comments/Remarks** |  |
| **Types of test cases** | Positive | - Covers intended or normal flow | | |
| Negative | - Cover unintended/exceptional flow | | |
| **Other things to note** | - Prioritise tests (i.e. define the priority of a test) especially when there is a strict timeline to complete testing  - If it is web application, include browser details (name and version)  - So that defect can be replicated easily  - Keep two separate sheets: “Bugs” and “Summary”  - “Summary”: summarise the test scenario  - “Bugs”: list all issues encountered during testing  - Provide an overview of the testing | | | |
| **Test Scenario** | | | | |
| - Define as any functionality that can be tested  - Gives a high-level idea of what is needed to be tested  - Example: “Check Login Functionality”  - Mostly derived from test artifacts like Business Requirement Specification (BRS) and Software Requirement Specification (SRS)  - Helps in an agile way of testing the end-to-end functionality  - Focuses on more “what to test” than “how to test” | | | | |

**Unit Testing, API Testing**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Unit Testing** | | | | | | |
|  | | |  | | | |
|  | | |  | | | |
| **API Testing** | | | | | | |
| - Software testing type that validates Application Programming Interfaces (APIs)  - Purpose is to check the functionality, reliability, performance, and security of the programming interfaces  - Mainly concentrates on the business logic layer of the software architecture  - Possible outputs of an API  1) Any type of data  2) Status (E.g. Pass or Fail)  3) Call another API function | | | **Base test case on** | | | **What to do** |
| Return value based on input condition | | | Define input, authenticate results |
| Does not return anything | | | Check behaviour of system |
| Trigger some other API/event/interrupt | | | Track those events |
| Update data structure | | | Authenticate effect on system |
| Modify certain resources | | | Access and validate modified sources |
| **Best practices**  - Group test cases by test category  - For each test case, include the declarations of the APIs being called  - Parameter selection should be explicitly mentioned in test case itself  - Each test case should be as self-contained and independent from dependencies as possible  - Avoid “test chaining” in development  - Take special note while handling one-time call functions  - E.g. “Delete”, “CloseWindow”  - Plan and perform well call sequencing  - Ensure complete test coverage, create API test cases for all possible input combinations of API | | | | | | |
| **API automation testing coverage** | **Discovery testing** | | | - Manually execute the set of calls documented in the API  - E.g. verify that a specific resource exposed by API can be listed, created and deleted as appropriate | | |
| **Usability testing** | | | - Verify whether API is functional and user-friendly  - Tests whether API integrates well with another platform | | |
| **Security testing** | | | - Testing includes what type of authentication is required  - E.g. Whether sensitive data is encrypted | | |
| **Automated testing** | | | - At the end, a set of scripts or a tool that can be used to execute the API regularly | | |
| **Documentation** | | | - Make sure that the documentation is adequate and provides enough information to interact with the API | | |
| **Possible types of bugs to detect** | - Fail to handle error conditions gracefully  - Unused flags  - Missing or duplicate functionality  - Reliability issues  - Difficulty in connecting/getting a response from API  - Security issues  - Multi-threading issues  - Performance issues  - API response time is very high  - Improper errors/warning to a caller  - Incorrect handling of valid argument values  - Response data not structured correctly  - E.g. JSON, XML etc | | | | | |
| **Differences between Unit and API testing** | | | | | | |
|  | | Unit Testing | | | API Testing | |
| Who performs the test | | Developers | | | Testers | |
| Functionality tested | | Separate | | | End-to-end | |
| Only basic functionalities | | | All functional issues | |
| Source code | | Developer can access | | | Testers cannot access | |
| Other tests | | Also involves UI testing | | | Only API functions tested | |
| Scope | | Limited | | | Broader | |
| When test is performed | | Usually before check-in | | | After build is created | |

**Integration Testing, Functional Testing**

|  |  |
| --- | --- |
| **Integration Testing** | |
| - Done to test the modules/components when integrated, verify that they work as expected (assume they work well individually)  - Does not happen at the end of the cycle, conducted simultaneously with the development |  |
| Integration test approaches:   1. Big-Bang   - Good approach for small systems  - Integrates all modules in one go   1. Bottom-up   - Start from lowest module, test it  - Gradually progress towards upper modules of application  - Integration continues till all module are integrated and entire application tested as a single unit  - Use *drivers* aka dummy programs, in case calling function does not exist (program that *calls* another)   1. Top-down   - Start from top module, test it in isolation  - Lower modules integrated one by one  - Use *stubs* aka snippet of code that accepts input and returns an output (function that *returns* something when called) |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

- https://www.softwaretestinghelp.com/the-difference-between-unit-integration-and-functional-testing/

- <https://www.janbasktraining.com/blog/integration-testing-tutorial/>

- <https://www.udemy.com/course/test-case/>

- <https://www.udemy.com/course/get-a-job-in-technology-without-experience/>

- <https://www.udemy.com/course/the-complete-quality-assurance/>

- <https://www.udemy.com/course/test-case/>

- <https://www.udemy.com/course/key-principles-of-testing/>

- <https://www.udemy.com/course/software-testing-business-analyst/?utm_source=adwords&utm_medium=udemyads&utm_campaign=LongTail_la.EN_cc.ROW&utm_content=deal4584&utm_term=_._ag_77879424214_._ad_437497333845_._kw__._de_c_._dm__._pl__._ti_dsa-1007766171552_._li_9062524_._pd__._&matchtype=b&gclid=EAIaIQobChMIpq-9g7O57AIVizUrCh3XogL7EAAYASAAEgLp7_D_BwE>

- <https://www.udemy.com/course/software-testing-business-analyst/> \*\*\*

- <https://www.testbytes.net/blog/heuristic-test-strategy-model/>

- <https://dev.to/hudsonburgess7/what-heuristics-do-you-use-when-developing-a-frontend-testing-strategy-4ah>

- <https://www.infoq.com/news/2009/10/testing_heuristics/>

- <https://www.udemy.com/course/software-testing-masterclass-from-novice-to-expert/>

- <https://learncsc.udemy.com/course/testerbootcamp/>

- https://www.udemy.com/course/the-complete-quality-assurance/