ICT Project Guidance

Quality Assurance

Version:

0.1

Author:

Sky Sigal, Solution Architect

## Description

<TODO>

## Synopsis

<TODO>

## Contents

[Description 1](#_Toc145232973)

[Synopsis 1](#_Toc145232974)

[Contents 2](#_Toc145232975)

[Synopsis 3](#_Toc145232976)

[Introduction 3](#_Toc145232977)

[Heading Level 3 3](#_Toc145232978)

[Heading Level 4 3](#_Toc145232979)

[Appendices 4](#_Toc145232980)

[Appendix A - Document Information 4](#_Toc145232981)

[Images 4](#_Toc145232982)

[Tables 4](#_Toc145232983)

[References 4](#_Toc145232984)

[Review Distribution 4](#_Toc145232985)

[Audience 4](#_Toc145232986)

[Structure 4](#_Toc145232987)

[Diagrams 4](#_Toc145232988)

[Terms 5](#_Toc145232989)

## Introduction

BOSSCARD/ RAID: Background [], Objective, Options, Scope[In/Out], Stakeholders [Users], Constraints, Assumptions, Risks, Dependencies, Decisions, Deliverables.

## Background

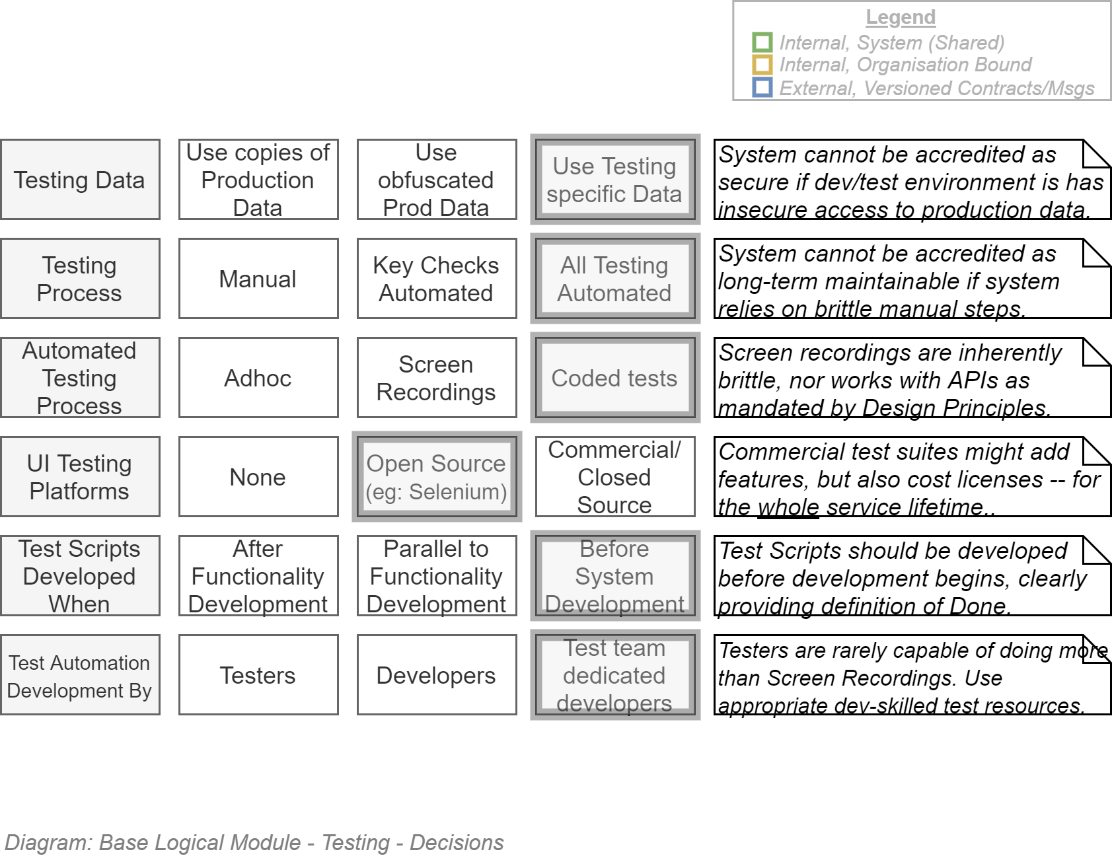
Traditional manual testing is perceived as a necessary task -- but also a notable bottleneck that impedes the rapid delivery of value to users and service providers.

## Constraints

### Testing Principles

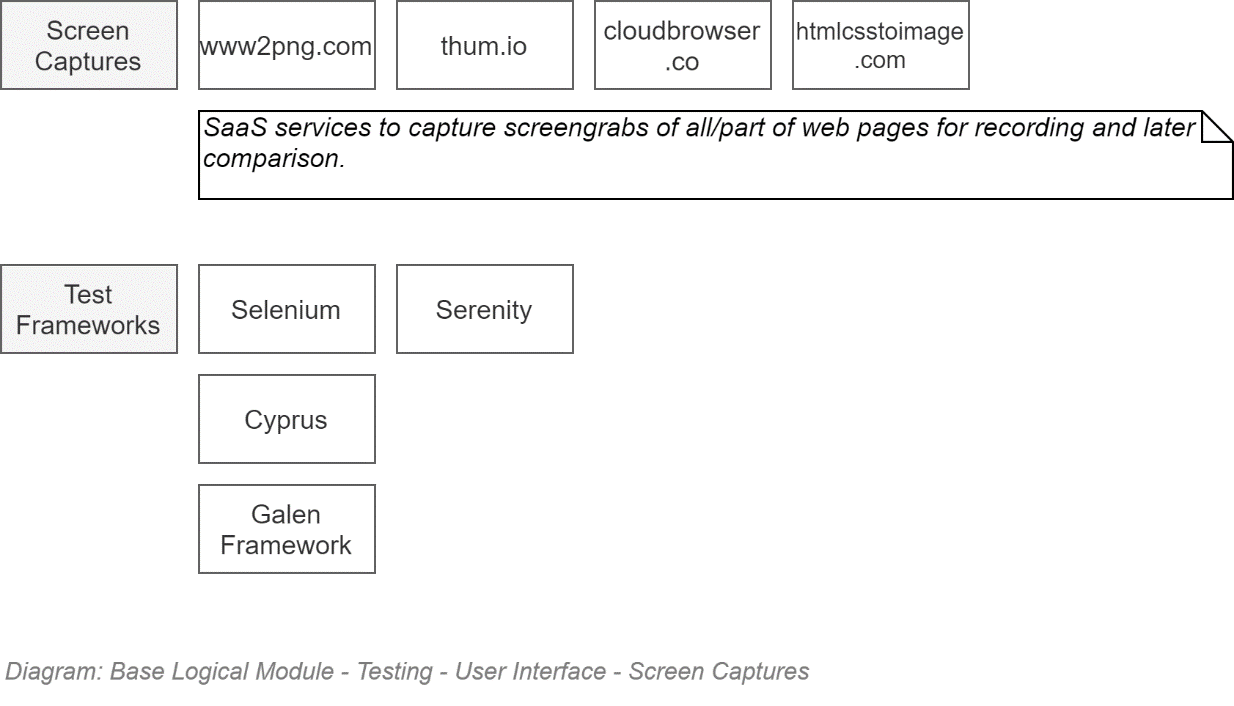
* **Automate all Tests**: Project Team Resources leave. Their experience with them. Tests encode this knowledge so that it can be run repeatedly, throughout the service’s lifetime.
* **No Development before Acceptance Tests are Defined**: a Work Item (User Story, etc.) without associated Acceptance Tests is incomplete. Without defined targets, developers can only guess as to acceptable targets, and end up only sometimes luckily correct. For documentation, velocity, and resource usage reasons, Developers must not accept to begin work on incomplete instructions.
* **Develop for Test Analysts before system Users**: developers see no real difference between developing code to meet the expectations of system users or tests analysts. So when they receive a work item (user story, etc.) they first develop automated tests according to the Acceptance Tests on the Work Item – and only then turn their attention to developing the system code to pass the coded acceptance tests run by the build and deployment pipeline.
* **Automatically test every check-in first**: test and reject errored code before it pollutes the main branch, forcing other developers to work around issues in the new code.

## Decisions



## Technologies

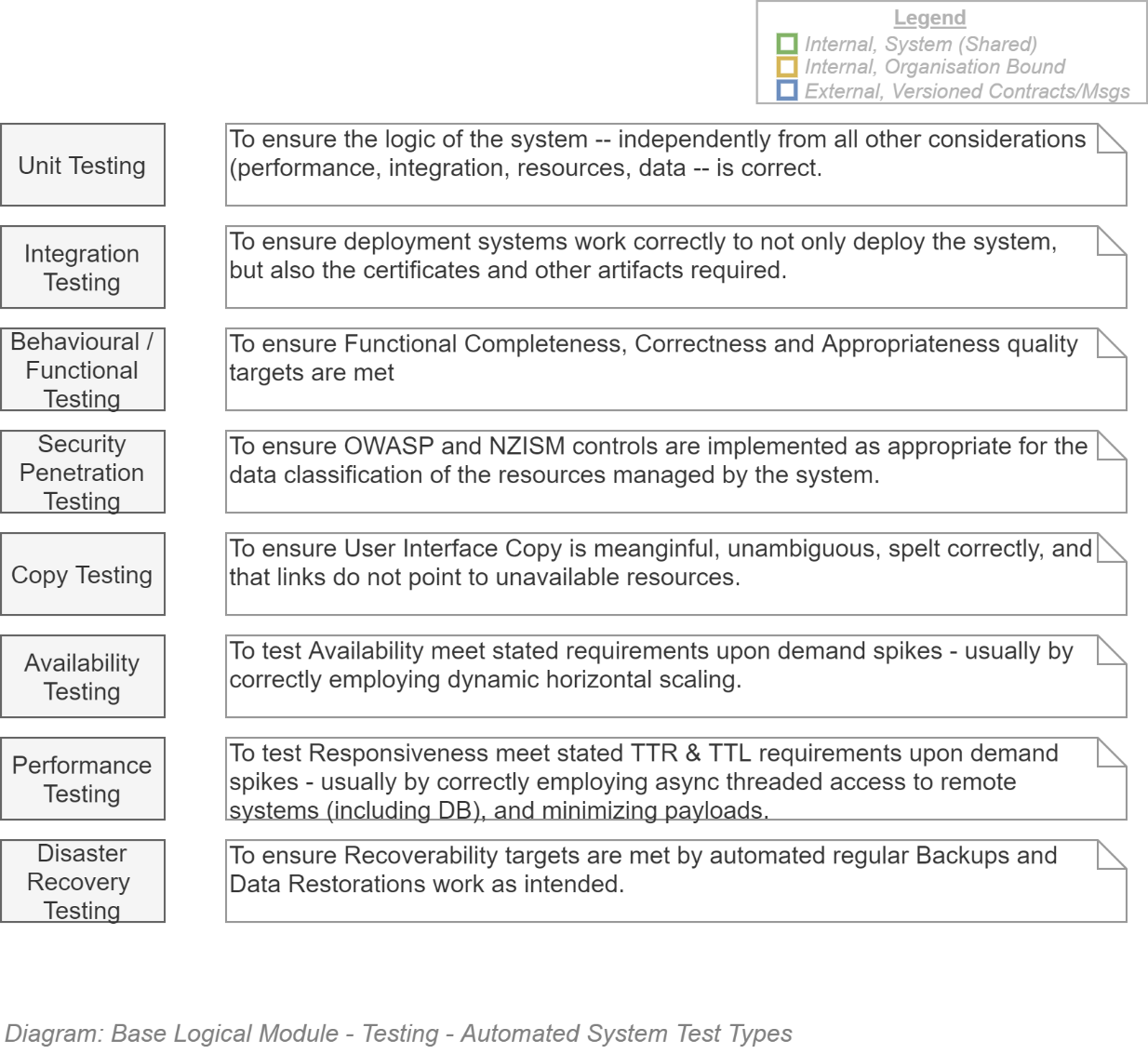
The following are a series of technologies to consider when developing the automated test framework.



## Test Types

Quality Assurance Tests do not *guarantee* that a system is error free – they just decrease the probability of there being remaining logical errors.

All contributing to decrease this probability, a series of different tests are used for different areas of the system:



Test definitions should be developed by different subject matter experts and developed by developers as much as possible – as described in the process below.

## Best Practice

Current best practices in testing depend on:

* Expanding the understanding of Team and delegation of responsibilities to the best role for the task:
  + Development of Test Scripts understandable by developers, *by Test Analysts* and their appending them to User Stories *before a developer accepts the work item*.
  + Exploratory Tests *by Testers* (usually the same persons fulfilling the Test Analyst Role, when there is time).
  + The Development of the Test Scripts into Automation *by Developers* – *and not accepting any work item by a Stakeholder Analyst that has not first sent the work item on for completion by a Test Analyst*.
  + Rapid, precise, repetitive testing of qualities and functionality being done by a robotic member of the team – *by the continuous delivery automation agent*.
  + Analysing the quality and potential early rejection to redo submitted code *by the continuous delivery automation agent* in a first instance, flagging the need for a secondary review by the development team lead if and as required.
  + Reports being developed automatically *by the continuous delivery automation agent*, on every new code commit, summarizing all qualities, made available to all stakeholders.  
    The design requirements of the reports are work items like any other – the actual coding of the report into the continuous delivery pipeline is done *by the developers*.
* Becoming a key part of the process to secure the system's data.

## Test Data

It is common in many organisations for testers to ask for copies of production data as the basis of testing.

**Risk:**The practice of using all or parts of production data is a well-known security lapse: backup data is still production data, just slightly stale.   
When exported from an ISO-27001 Stage 2+ certified environment, available for viewing, copying and publishing from most team members (generally inclusive of most testers, BAs, developers), it's just slightly stale production data, with no access controls, access auditing or attribution.

Develop test data based on:

* desired outcomes, as per Stakeholder Analyst work items
* known issues discovered by automated data migration (via APIs), exploratory testing, or user defect reporting

### Data Migration Testing

It was unfortunately common that Data Migrations were done at a database level, using ETL or similar processes. This approach is considered poor practice as it bypasses all validation and workflow logic, requiring an additional expensive and time-consuming level of testing at the database tier.

Perform Data Migrations via the System's APIs, where the data is validated, raising a set of exceptions that become the basis of new work items, prioritised alongside all other work items.

## Process

A legacy approach to testing involves both

* developing test scripts in parallel to development and
* manual testing -- omitting to use automation as a team member.

The legacy approach is a flawed process due to several factors, including – but not limited to – the following:

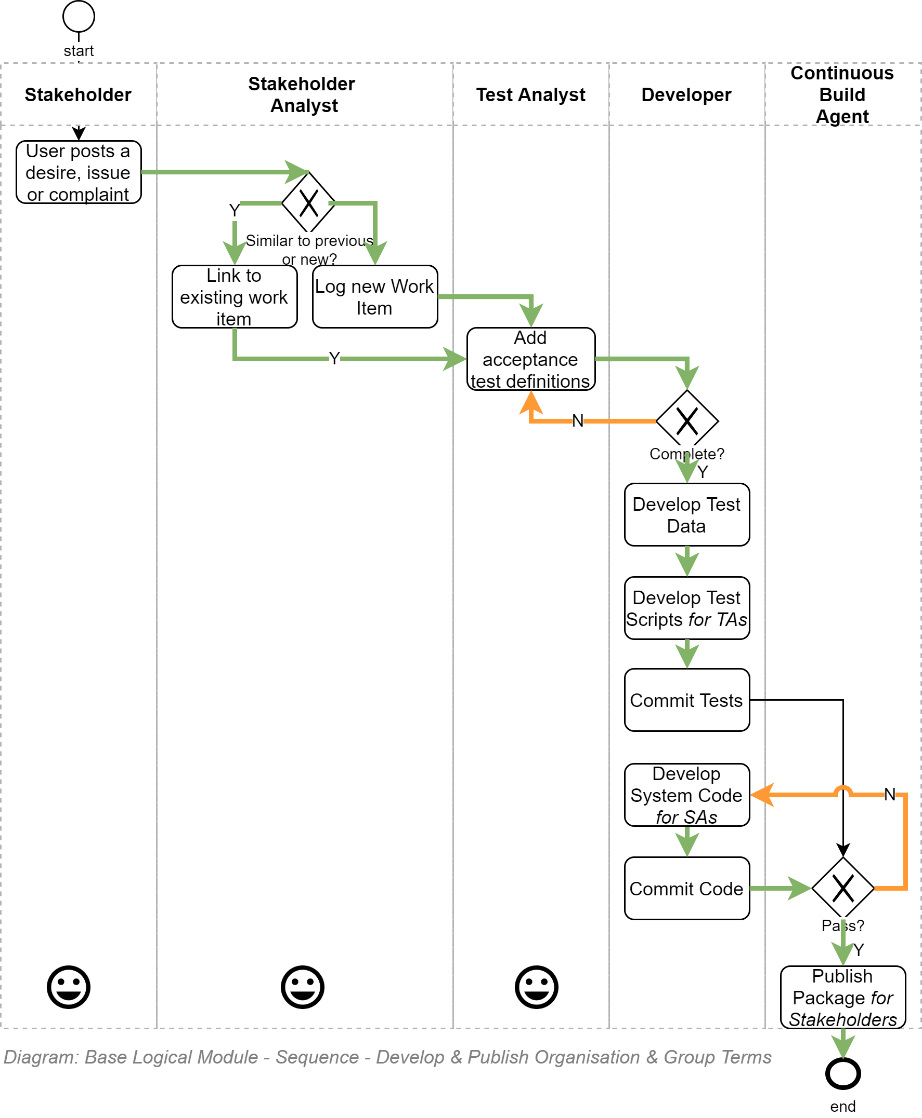
* By passing Stakeholder Analyst Work Item definitions (e.g.: User Stories) to two different groups (Testers and Developers) to interpret independently, reveals the potential for ambiguity that exists in plain language. In the era of distributed development and multiple cultures, this effect is compounded by errors in translation.   
  The result is that when tested there is a 20-30% chance that the both groups did not understand the work item in the same way – which by default is equal to a Failed Test.  
  20-30% failed tests, and ensuing re-work, is a certain recipe for failed deadlines.
* The traditional approach where Testers are used to develop scripts to test automation by manual tests is …simply backwards in terms of time and precision. Use automation to test automation at the speed and precision of automation.
* The slight variant of the traditional approach where Testers develop manual test scripts and then use recorders to record these actions to replay them as automation is not much better. Instead of testing the logic of systems, they test the user interface of systems – the layer with the most churn over service lifetime -- and are therefore both brittle and highly expensive to maintain in a functioning state.
* The traditional approach – whether automated or manual – is highly associated to the testers desktop, leaving the automated build pipeline as an underused afterthought.

The alternative approach is to solve both problems in one go by:

* Ensuring that Testers are tasked only with Analysing work items to develop and complete them with appropriate Test Data and Test Definitions.
* Ensuring that Test Analysts analyse the Work Item (i.e., User Stories) Definitions in sequence, between the Stakeholder analyst and Developer, and not in parallel. This removes the chance of differing opinions as to the actual intent of the Work Item’s plain language definition.
* Ensuring professional Developers are used to develop automation, rather than testers[[1]](#footnote-2).
* Ensuring the continuous build and test engine is used as a trusted team member to do what it is good at: automating builds, tests and packaging.

The outcome of working this way is that the developer’s target is reduced to a single set of objectives -- rather than split between trying to please two different expectations that may not match – proven quickly and reliably with appropriate automation.

The higher value process to be used on this project is shown below.



## Examples

### Desires

Desires are what end users or stakeholders put on the user feedback log.

They are often unclear, incomplete, cover more than one distinct concern, may be a duplicate of previous desires posted by others which have not yet been finished or even prioritised to be worked on.

Effort is required to analyse a posted Desire and develop appropriate Work Items (i.e. User Stories and/or Tasks) from them. And link them together so that users can remain informed as to progress – if any -- on their comment regarding the system.

A Desire may simple look like:

“I’ve signed in and it all works – but often times when I go away for a weekend or longer and return, when I try to get to my dashboard again, I get an error. But I can get to the Home page.”

The above example of a Desire appears to cover multiple domains – it’s seems to include basic authorisation, session timeout issues, a lack of graceful error messaging to inform users what to do next.

### Work Item

A Stakeholder Analyst may convert the above desire into appropriate Work Items, using both SMART guidelines, as well as using the well know User Story format (For <valued reason>, As A <role>, I desire <outcome>).

Note:  
There are two well-known Agile user story formats, one starting with the <reason>, the other starting with the role. Although Stakeholder Roles are important to please I highly recommend starting Work Item definitions stating the <valued reason> first -- to ensure the work item is clear for all as to what *shared* *value* the effort (i.e. *shared cost*) is supposed to deliver.

*For Security Reasons*  
*I desire Authenticated Users*   
*Be able to view their Dashboard*

### Acceptance Tests

Feature: Dashboard Access

# The first example has two steps

Scenario: Maker starts a game

Given an @UnauthenticatedUser  
When the user requests ‘/Home  
They receive a response with an HTTP code of 200.

Given an @AuthenticatedUser  
When the user requests ‘/Dashboard’  
They receive a response with an HTTP code of 200.

Given an @UnauthenticatedUser  
When the user requests ‘/Dashboard’  
They receive a response with an HTTP code of 401

## Functional Testing

**Tip:**  
Unless one is designing a like for like service replacement with no functionality improvement or change – which should never be the case – the service is – by intent, design and implementation – a different system, who's functionality should not match another system. It should have coherent functionality in its own rights.

### API Service Façades

The following are a high-level list of the Tests to ensure are covered.

**Note:**  
For succinctness' sake, the following Action terms are used in the listings that follow:

**List:** the full objective of the test is to ensure GET-ing "Paged, Filtered Lists of Synopsis records summarizing the record itself".

In many cases the List responses will require Filtering by a specific Group or User.

**View:** An HTTP GET a single record (not a Synopsis of the record).   
Create: an HTTP PUT of a package containing the information required to create a new record, and potentially child records.

**Update:** a POST-ing of a Message containing the information required to update a record, and potentially create or update child record at the same time.

**State:** a specialisation of Update, updating a single field (e.g.: Approved/Enable/Disable/soft Deleted, etc.)

**Delete:** a special case of State.

**Order:** a special case of State, setting the natural order of a Collection (by Date, Name, etc.)

**Upload:** a special case of PUT which includes Media requiring Malware detection and metadata development.

**Assign:** PUT/POST/DELETE operations to associate Permissions and Roles to Groups, Users and Resources.

APIs to test:

* Search Service: List
* Configuration Service: List, View
* Settings Service: List, View
* Diagnostics Service: List, View
* Error Record Service: List, View
* Components Management: List, View
* Modules Service: List, View, State (Available/Disable)
* Session Service: List, View
  + Session Operations Service: List, View
* Identity Providers: List, View
* Users: List, View, Create, Update, State (Enable/Disable/soft Delete)
  + User Identities: List, View, State
  + User Login History: List, View
* Permissions: List, View, Create, State (Enable/Disable/soft Delete)
* System Role Management
  + System Role Permission Assignment: List, View, Create, Update, State (Enabled/Disable/soft Delete)
  + User System Role Assignment: Assign, Remove, add/remove sub-Permission
* Group: List, View, Create, State (Disable/Enable, soft Delete)
* Group Role Management: List, View, State (Disable, Enable, soft Delete)
  + Group Role Permission Assignment: List, View, add/remove sub-Permissions
  + User Group Role Assignment: Assign, Remove, add/remove sub-Permissions
* Media Management: List, View, Assign, Upload
* Media Collection Management: List, View, Create, Update, State, Order
* Media Role Management: List (by Group), View, Create, Update, State
  + User Media Role Assignment: Assign, Remove
* Workflow Management: List, View, Create, State
  + Workflow Step Management: List, View, Create, State
  + Workflow Step Role Assignment: Assign, State

Applicable to all the above test:

* All the Web API tests MUST be executed by automation in the Continuous Delivery Pipeline.
* All the above Tests MUST be executed with test-specific produced data. The use of production data in any shape or quantity is a security risk that will not be tolerated on this project.

Important:  
Note that almost all of the above functionality is provable by User Interface Testing (see below).   
For the sake of expedience, only develop API Tests for functionality that is not evidenced by a Service Client UI.

### Web User Interface

The above list of functionalities is tested again in the interface:

* ***Almost All*** the above API functionality MUST be available via the supplied User Interface in some form or other (doesn't automatically have to be pretty). Specific differences include:
  + the interface generally does not need to form Projections of Listings of Synopsis objects,
  + views don't show all values that an API may return (for example, the UUID of the record, etc.)
  + some state changes are intended for remote API clients or internal workflows to invoke and are maybe not intended to be surfaced on a UI (to be determined on a case by case basis).
* All the Web Interface tests MUST be executed by automation in the Continuous Delivery Pipeline.
* All the above Tests MUST be executed with test-specific produced data. The use of production data in any shape or quantity is a security risk that will not be tolerated on this project.

## Quality Testing

Refer to ISO-20010, ISO-20012 and ISO-20022 for the types of Qualities to test the system for.

### Prioritization of Quality Testing

It is my professional opinion to ensure that qualities testing should proceed in the order listed presented below.

#### Dependencies

The test order places a dependency on security and infrastructure expertise beyond the professional boundaries of functional testers.

Testers are not expected to test these qualities themselves – but they are expected to be intransigent that testing of these qualities are performed by automation within the continuous deployment pipeline and reported to them before they proceed with evaluating the reports of other test groups (including functional testing).

#### Order

* Security: the service MUST NOT be made available if there is any known unmitigated security vulnerability. Specifically:
  + All connections between components MUST be encrypted when technically achievable.
  + All data stores (production and backups) MUST be encrypted.
* Privacy: the service SHOULD NOT be made available if Privacy Principles. Specifically:
  + The system MUST NOT request more PII than required to function.
  + The system MUST encrypt database columns containing PII information.
* Accessibility: the service MUST NOT be made available if the user interface is not accessible to visually impaired users.   
  The service MUST NOT be made available if the user interface has not already made to allow for localisation of text and copy in multiple languages.
* Usability: the service SHOULD NOT (but MAY be formally Decided) be released for preview only before effort has been made to meet Usability standards.
* Availability:
  + the service MUST NOT be made available for users to add data if Backups and Restoration are in effect.
* Resilience:
  + The service SHOULD NOT be made available for users before DR requirements are met by infrastructure being fully developed by automation.
  + The service MUST scale horizontally to handle peak demand.
* Maintainability:
  + The service SHOULD NOT (but May) be made available to users without effort to summarise the maintainable state of the code (summarised as complexity metrics, etc.).
* Functionality:
  + Only when the previous qualities are met should the system be tested for functionality completeness and correctness.

## Summary

Tester Analysts have less manual testing to do – and much closer involvement with Stakeholder Analysts on a workitem by workitem basis. This approach provides them more time to think of how to test the system.

In addition, rather than being last to be considered, Testers become developer clients with precedence over all other (eg: business) Stakeholders, as it is their code that is used to test the other stakeholders' requests.

Appendices

Appendix A - Document Information

### Versions

* 1. Initial Draft

### Images

[Figure 1: TODO Image 2](#_Toc144995112)

### Tables

[Table 1: TODO Table 3](#_Toc145048484)

[Table 2: TODO Table 2 3](#_Toc145048485)

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Terms

Refer to the project’s Glossary.

##### IT

: acronym for Information, using Technology to automate and facilitate its management.

##### ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. IT is a subset of ICT.

1. It takes approximately 5 years to make a competent system developer – developing resilient automation instructions by Testers having to undergo a career change from manual testing has low value. [↑](#footnote-ref-2)