Testing

We aim to build quality into our software from the outset to build confidence in the quality, reliability and maintainability of our code base. To achieve this we should all be aiming to follow Test Driven Development and Behaviour Driven Development practices. Apart from the ease of maintaining and extending our code base, we use code coverage as a measure of the quality; we have a code coverage target of 80%. It is worth stating that code coverage is a single metric that represents an overview of one aspect of your test coverage; what percentage of statements in your code base have executed as part of your test executions. Unfortunately, code coverage does not ensure good quality code as it can be easily achieved by writing non-failing tests (a test without an insert wrapped in an exception handler block). To achieve the real target of quality code we should aim to implement good quality tests that provide true test coverage on the functionality being developed.

The following sections categorise the various types of tests we run and defines a simple scope for what you should be putting into the tests in that area.

Unit

Unit tests are executed whenever we perform a build on an application, they test the smallest components of the system and all dependencies should be mocked.. When writing these tests you should adopt the Arrange Act Assert test approach.

Wherever possible strive to have a single, distinct, clear action call backed by a single assertion. For example rather than checking the size and individual contents of a return collection in multiple assert statements prefer to build an expected collection in the Arrange section and assert the expected and actual collections are equal. There will be times, especially with legacy code, where this is impractical; in these case still aim to reduce the number of individual assertions whilst still covering all the test conditions.

Integration

There is often a need to ensure that functionality across a number of individual components (eg, classes) functions correctly. Although these are named integration tests they should only cover components in a single repository and, where possible, not depend on any external dependencies. As with unit tests, dependencies outside of the tested components should be mocked. Once again, we these should be implemented using the AAA pattern.

Acceptance

The acceptance tests are executed as part of each build to ensure that changes are successful and existing functionality has not been broken. These tests will provide and consume data on the build machine. Ideally all external dependancies will be mocked but in the cases where it is not possible the acceptance test should ideally clean up after completion. These test should all be "black box" tests and test the public interfaces of the application/service. We use Gherkin language to run all acceptance test under cucumber. When picking up a new story you should find all the "business" acceptance tests fully stated in the story with realistic measures; however, you should also consider if you should be adding NFR (Non Functional Requirement) acceptance criteria to these as part of the development.

Gherkin

Tests are divided into Features, each feature can contain multiple Scenarios where a scenario is a specific test. Scenarios are written as English statements using the main keywords of "Given", "When" and "Then"; written in this way they map directly to the unit test AAA principle. To allow for fluent tests you can also use the joining keywords of "And" or "But". Where possible we strive to have a single, distinct, clear action (When clause) and as few assertions ("Then") clauses as possible. It is expected that Gherkin test should read as plain English and not require knowledge of the underlying implementation.

All interfaces, published in api-docs, must be covered by acceptance tests along with all acceptance criteria specified in the associated story.

Automated

Automated tests are used to ensure that a deployment of the code has been successful. They are akin to health checks in the system but are only used as part of the deployment; often the health checks will be included as part of the automated tests. We will run these tests on all deployments so they should not change the state of the system or negatively impact the performance of the system; we do not want hidden state or data changes to affect an environment especially production. Typically, there will be a small number of automated tests and they are there to ensure the public interface is fully operational.

Regression

Our aim is to move all regression style tests into acceptance tests for each component to reduce the cost of maintenance and improve our level of confidence with the overall system state. However, we have a number of larger applications that do not have adequate acceptance test coverage and we therefore rely on a number of different regression test packs to provide confidence. These regression tests should be maintained whenever the underlying software has a functional update but we should not need to develop/extend the regression packs for newer software.

Performance

Currently we have three performance test suites; Full site, aggregation only and services only, they all use Gatling to run tests at 2x peak load. Performance tests are executed against a performance environment that is a full replica of the production environment. For full site and aggregation test we mock out calls to our providers using cached/mocked provider results. Further details of performance testing can be found in the following pages:

Performance Testing

How To Run A Performance Test of TSM Agg

Manual Testing

In addition to all the above automated testing practices we also run manual testing to ensure that developed features work in any foreseeable scenario. Along with ensuring the base functionality in the acceptance tests manual testing is an exploratory testing phase where we would try to "break" the system by using it in usual or extreme ways.

Mutation Testing

Mutation testing is a testing approach that we have considered but have not yet adopted into our testing strategy. Basically, mutation testing takes a working tested code and mutates the code with the expectation that this will cause unit tests to break.