Financial Services Client

On-demand pricing trade calculation

Problem

The client had an internal, sales facing website which allowed users to define a set of pricing requests and get back a set of risk and scenario numbers for them. These requests could be expressed in absolute terms, e.g. a strike of 0.76 or using domain specific values such as a 25d put.

They wanted to be able to see the impact of proposed changes to both market configuration[1] and the software which was used as well as get validation during the CI process that, where none were expected, there were no differences[2]. The underlying software stack had unit tests for known use-cases, but these were created for[3] specific use-cases[4] rather than being guaranteed to be the cases which were being used in production at the time.

The team consumed libraries from multiple other internal teams. For some of these other teams, their primary client was the pricing system and as such, these teams wished to be able to get prior warning of the impact of their proposed changes before formally crystallising their releases.

Note that it was possible to run a self-contained stack locally.

[1] E.g. changing the way a yield curve was constructed
[2] This dual purpose is very common within our customers' requirements
[3] And subsequently updated when differences were found
[4] Which were known at the time of code creation. This frequently led to there being unit tests using configurations which were no longer in actual use by the end system and no unit tests covering configurations which were actually being used.

Solution

The solution here was a multi stage process:

1. Define the range of inputs which should be tested for, e.g. strikes - {1.2, 1.4, 1.6, 25d, atmf, spot}, ccypair - {GBPUSD}, stype - {put,call} etc. This was done in conjunction with the quants[5], the system designers and the end users.
2. On a regular basis, expand the set of inputs to the set of actual pricing requests based off of the market for the reference date, price accordingly and then store for subsequent use
3. When tests are run, read in the test material generated from the previous step, execute against the candidate web service and compare the results

Note that the test material was stored in the same source control repository as the code for the project. This was to ensure that the code and expected numbers could be updated in a single atomic unit.

A console app was written so that the tests could be run both as part of the CI process and locally. When running in local mode, it published its results to Conical so that developers etc could easily understand where the changes were and then share with the appropriate colleagues for approval prior to code being checked in.

The CI process was updated to run the generated test material in a strict pass-fail mode. This meant that a user had to run the tests locally / on a branch and update the set of expected results (see later for details on how that was achieved painlessly) in order for their commit to be successful.

Each step is defined in more detail below.

[5] The team responsible for performing the actual analytics

Input definition

In order to reduce the need to enter in large combinations of trades, a simple template format was defined along the lines of 1 or more of the following block types per test[6]:

```
# Single value options are treated as fixed
TradeType: FXOption
CurrencyPair: GBPUSD
Type: Put,Call,Straddle
Strike: 1.1, 1.2,1.3,1.4,spot,atmf
Expiry: 1w,1m,3m,6m,1y
```

In the above instance, 90 (3 x 6 x 5) pricing trades would be generated[7].

These would then be grouped into logical units, so that the 'FX / Flow / Europe / 2022-05-02' test would contain a large number of different currency pairs etc[8].

[6] The actual format allowed for more complex and allowed for the ability to select n of the various input options to prevent the combinatorial explosion of very similar tests being generated.
[7] See #3, the actual format would allow for 'select 3 of the following options', e.g. Strike,3,1.1,1.2,1.3,1.4,spot,atmf etc.
[8] The actual choice of this categorisation was a matter of some debate between the various teams and they settled on what suited them best.

Test generation and storage

A weekly CI job was set up to generate test material for a new reference date using the previously defined templates. This CI job would create new material for the reference date and mark older material for removal so that the total volume of test material was kept manageable. This CI job could also be manually triggered if additional material needed to be created off-schedule[9]. This would generate a change request which could either be automatically committed[10] or could be emailed to the development team for approval.

This material was generated using a custom console app to take the above templates, expand them out and send them through to the pricing service and then subsequently store both the expanded out inputs and the expected results.

This whole process made it very easy to ensure that the test material being used to validate the system was up to date and that adding additional test definitions was as simple as modifying the test templates and then they would automatically flow through.

The choice was made to store the data in source control so that code changes and the associated impact of those changes could be processed in a single atomic unit of work.

[9] Typically when a new feature had been added to the pricer and people were keen to have the material ASAP.

[10] When operating in this mode, prior to the commit being performed, the generated material was run again so that the build wasn't going to be left in a broken state over the weekend if something went wrong.

Test running

These tests were run by a custom executable which imported the inputs and then called out to the pricing service. These results were then compared against the set of expected results using the BorsukSoftware.ObjectFlattener and BorsukSoftware.ObjectComparison libraries. Where the numbers matched, no payload was necessary and the test was marked as a success. Where there were differences, a difference payload[11] was generated and the set of expected results were stored along with a mapping file[12].

Once all of the tests had been run, then an additional file containing the details of all of the tests whose expected results should be updated if the updated code / configuration should be accepted was generated and stored. This file could subsequently be used by automated tools to update all the TMR[13] without needing to manually update the result files for multiple tests.

These tests could be run locally by developers with the results stored locally on disc or they could also be published to Conical when they were being run as part of the CI process or release confirmation. Conical mode allowed for users to be able to share the impact of proposed changes with their user-base / library contributors for easier confirmation and therefore quicker release cycles.

Note that this ability to run locally was also available to developers in supplier teams[14] assuming that there were no binary compatibility issues so they even run their tests without needing to get the pricing systems team involved[15].

[11] A Json document detailing a summary of the number of inputs which matched / didn't match etc along with a detailed drill down of what didn't match for any given input.
[12] This file specified what files should be replaced in the TMR with which saved down file in the case that the new results were accepted. Typically, this file was simply 'expectedResults.xml => additionalFiles\expectedResults.xml' but was flexible to allow for more complicated test definitions in the future
[13] This worked by reading in the previous files, copying them over and then generating a change request which could be submitted to source control in the usual way.
[14] Think 'quants' here
[15] This ability help to encourage a change in culture from the suppliers as it was possible to run the full set of tests in a timely fashion so their libraries could be accepted and released to the end clients much more quickly.

Variations

The general approach detailed here has been very common across most of our clients, there are slight variations based off whether the analytics are being called via a web service or where there's no inherent web service, but instead the functionality is exposed through a set of binaries[16].

Note that almost the entire case study holds true if one's analysing results for a custom scenario processor, the only differences would be around:

1. How are scenarios defined (usually using a local template)
2. How to express the payload, by trade and then scenario or by scenario and then trade[17]

A similar process also applies when dealing with multi tiered risk systems[18]. By testing each layer and having confidence that it behaves as expected, testing of subsequent layers becomes easier. Of course, one can alway start by testing the top level[19] and then expanding downwards as appropriate.

The common portion is always to start with the question 'what will make the release process for our customers simpler and more reliable?' and then to go from there.

[16] At this point, typically a 2nd custom app to call the actual binaries in-process is created. This 2nd app is then called by the 1st, once per actual test to be run. This is so that additional metadata about the run, e.g. memory consumption, can be generated and stored down with the results
[17] Conical allows users to upload multiple files so both payloads could be delivered for subsequent analysis
[18] One could imagine having a layer which can calculate risk for a single trade under a single market state, and then having additional layers above which provide portfolio level functionality, scenarios, 2nd order risk functionality etc. etc.
[19] The one consumed by the end user