Mobile app development
UX testing

Problem

We had a client who was developing a new cross platform[1] mobile app and they were looking for a way to confirm to the various stakeholders that the app was performing as expected.

The end user had expressed a preference for using AWS's device farm (link) to perform the end user testing and as such, this was to form part of the solution.

The proposed solution was also required to help the developers fix problems as they arose, so it was a requirement to be able to run the tests locally.

Note that this testing was in addition to the usual set of unit tests.

[1] iOS and Android

Solution

At a high level, the problem was split into 3 stages:

1. Defining and writing the set of tests to be run
2. Automating the orchestration of running the tests
3. Displaying the results / allowing them to be consumed in an easy fashion

Defining the set of tests

For this, the end client had a series of user stories and as such, these would form the basis of the tests. Additional tests were added to ensure that error conditions were handled gracefully etc.

Given the existing skill set of the client was a combination of .net and python, a choice was made to use appium tests written in python. These are natively supported by AWS within the device farm and can be easily run locally.

The following design decisions were taken for the tests:

- Pytest would be used
- A simple class hierarchy was set up to contain all of the boilerplate code, thus meaning that all the test author needed to do was interact with the appium instance
- Tests would take screenshots[2] throughout their runs for ease of subsequent analysis

As pytest was used, the tests could be run locally, against either the official build from the build server or a debug build on a developer's machine. A decision was taken to use the '-rA' flag[3] consistently so that it was trivial to subsequently parse the logs.

[2] Appium natively supports taking screenshots and the AWS device farm has the ability to report any saved screenshots as part of its job artefacts. A 'by convention' to naming was taken so that when results were later uploaded to Conical, it was trivial to automatically attribute a screenshot to a given test

[3] This expands to showing a summary for all tests at the end of the run (link)

Automated running

The client was using Azure devops to orchestrate their BTR process. As such, an additional step was added to the pipeline[4]. This step did the following:

1. Download the main build
2. Zip up the tests to run
3. Launch the AWS job using a custom tool (nuget)
4. Wait for the AWS job to complete using a custom tool (nuget)
5. Upload the results to Conical using a custom tool (nuget)

The main complexity that occurred was the fact that the pipeline stage is synchronous whereas calls to the AWS device farm are asynchronous. Given that there's no guarantee that AWS will have the requested resources[5] to hand immediately, there can be a significant[6] delay involved.

Note that because developers were able to run the tests locally, whether the full set or a subset[7], they were able to verify that their changes didn't have a negative impact on the expected end user experience prior to checking their code in. If a change was detected, then they could either update their code or update the test accordingly, depending on what was required.

[4] Always run on main, optional for PRs / branches. This was done for a couple of reasons, cost - at the time of writing, the device farm costs US$0.17 per minute - and timing - the tests took ~20 minutes to run and the developers didn't want to be delayed when it wasn't strictly necessary.
[5] A choice was made to use a range of static device pool configurations to reflect the client's expectations of the set of devices on which their app would be used.
[6] The timeout was eventually set to 90 minutes to prevent failures due to resource contention. A manual pipeline step was added to publish the results to handle cases where the job timed out from a devops perspective but eventually completed on AWS.
[7] Using pytest's filter functionality when running from the command line

Uploading / display results

When uploading the results to Conical, the following decisions were taken:

- Per AWS job:
    - each device would be treated as its own test run set
    - an evidence set would be generated within a device specific prefix[8]
- A dashboard with a URL linked to the build number would be generated per pipeline to provide a shortcut to all of the available testing evidence[9] for the pipeline.
- Each test run would contain the appropriate portion of the pytest logs and the screen shots generated during the run
- Although the screenshots would be uploaded to Conical as additional files, the ‘device video’ which AWS provides as part of the service was not. Instead, an external link to this video on AWS’s console was provided[10].

With this set up, the client was able to provide their stakeholders with full visibility on the state of the app under development which significantly aided in getting their approvals for releases.

[8] This prefix was a function of ‘OS family / manufacturer / device (OS version)’. This made it very easy to spot when a problem occurred whether it was a function of the app being tested - at which point, it would affect all devices - or whether it was device specific.
[9] To this end, the results of the front end unit tests were also uploaded to Conical.
[10] This decision was taken as it was very rare that the video was useful and the additional storage requirements / upload time etc. weren’t seen as worth it.

Summary

By automating the running of the ux tests and displaying them in a way which was easy for non-technical stakeholders to consume, we were able to reduce the time it took for new releases to be signed off, thus improving time to market.