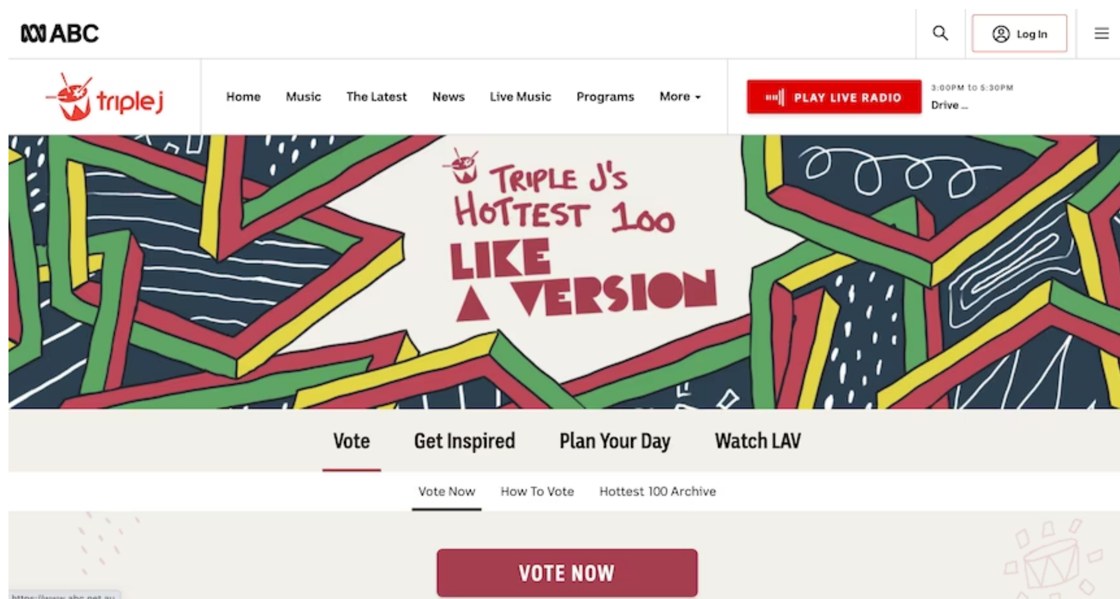
How we tested triple j's Hottest 100 of Like A Version and left nothing to chance

By Richard King

Posted Tue 27 Jun 2023 at 2:00pm, updated Wed 28 Jun 2023 at 4:10am



triple j's Hottest 100 is one of the ABC's most treasured institutions, with a history stretching back decades, so when I was given the job of testing the voting system for the first ever Hottest 100 of Like a Version, there was a huge responsibility to get it right and ensure it performed seamlessly.

In this blog post, I'll take you through how we built automated tests ahead of time to check that critical elements of this project would work correctly as the event progressed.

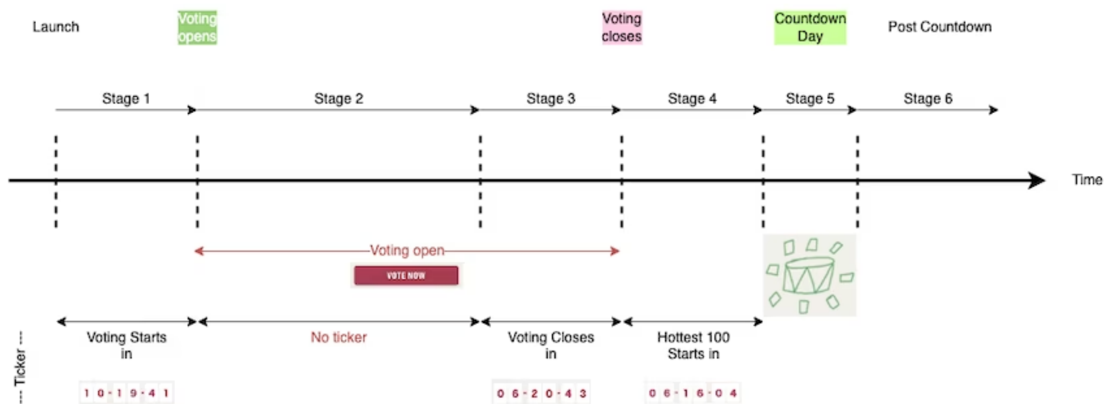
The stages of the vote

Voting in the Hottest 100 of Like a Version is currently underway, but if you had a time machine and went back a few weeks, instead of seeing a "Vote Now" button on the triple j website, you would see a ticker counting down to the start of voting:

Likewise, if you went forward to July 11, you would discover voting had closed and the ticker would be counting down to the day of the Hottest 100 of Like a Version.

Go forward to the day of the music countdown – July 15, put it in your diary – and you would see that the countdown itself was taking place.

The web application ("the app") for this voting system is configured to adjust itself as time advances and, like a chameleon, change its look and behaviour as it moves through the six stages of the event.



Requirements

These were the two critical elements of the Hottest 100 of Like a Version:

- the "Vote Now" button (and the link to the voting site) must only be shown during stages 2 and 3;
- the ticker counting down to both the end of voting and the day of the event itself must only be shown during stages 1, 3 and 4.

There were other important aspects of the app that also needed to be included in the tests:

- voting requires that a user be logged in;
- a user cannot vote twice with the same email address;
- once a user has voted, a "thank you" page should be displayed and they should not be allowed to vote again;

- no defined components on a page (there are numerous pages for each stage) should be missing;
- no component should carry over into a later stage if not defined for that stage.

The challenge in building these tests is verifying that the behaviour has been pre-programmed correctly -- for instance, we don't want to discover that no one can vote when the date and time for voting to open arrives, especially since triple j is announcing that voting is opening at that same time.

Similarly, we don't want to discover that a user could vote a day before voting officially opened or a day after voting officially closed.

This required that we not only check that the dates and times at which a stage should start and end are correct, but that the contents of the pages of the six stages are also correct. Not everything could be programmed from the get-go. For instance, we do not know exactly what certain pages will show since not all of their content is hard-coded – some content is coming from a Content Management System (CMS) instead -- but we can check that the components defined for the pages of each stage are correct.

As mentioned above, checks were made not only for what should be on a page, but also for what should not be on a page.

Of course, we could test these six stages manually and there was, at least initially, some manual testing of the start and end dates and the times of the stages by changing the system date and time of a test machine.

This was not sustainable and would become tedious and error prone to repeat as more and more features were added to the app -- not to mention how tiresome it would have been to repeat all stage start and end checks manually when the countdown was delayed by a few weeks.

Capturing the requirements as Executable Specifications

This blog post will focus on the two most critical requirements from here on:

1. the stage start and end dates are verified to be configured correctly; that is, the Vote Now button and the ticker, when shown, is counting down the days, hours, minutes and eventually seconds correctly for the defined stages;
2. the navigation items and the content of the pages for the various stages is correct and nothing is missing or still included from the previous stage.

At this stage, I will introduce Gherkin as the language in which we captured the requirements as "executable specifications". You will need to use an implementation of Gherkin, such as

Cucumber that is compatible with your testing framework(s). There will probably be a choice of Cucumber implementations for your testing framework. From here on we will use Cucumber as the generic name for a tool that allows you to write requirements in the Gherkin language.

Cucumber provides a simple Domain Specific Language (DSL) in which to write the tests at a high level, better understood with a simple example:

Cucumber Scenarios are collected together in Feature files. They typically have a Given, When, Then structure, but this is flexible and not enforced (the keywords can in fact all be asterisks).

Cucumber allowed the tests to be designed and written early on, well before the first cut of the app was available for testing. Other benefits of capturing the requirements as scenarios is that it provides a reference point for discussion before writing the actual test scripts and if needed the tests can be restructured at this stage. When returned to for the next Hottest 100, the scenarios will provide documentation on the previous competition's test scenarios.

Sitting behind the Cucumber feature files is the Javascript code to drive the app running in a browser. Another benefit of starting with Gherkin/Cucumber is that the choice of testing framework (Webdriverio, Cypress, Playwright, etc) can be deferred for a time (and even changed later on if one framework falls out of favour or another gains useful features).

In the case of triple j's Hottest 100 of Like A Version, the requirements drove the choice of testing frameworks to use:

- Cypress to do the time-travel checking (no other testing framework provides the ease and level of support required for simply changing the browser date and time); and

- Webdriverio to do the page content and visual layout checking on a variety of desktop and mobile browsers (neither Cypress or Playwright supports the extent of browser/OS/device combinations required). The ABC supports its products and apps on browsers and devices where the audience numbers are greater than 1% of the total.

The source of the Cucumber "executable specifications" was the product documentation for Stages 1 to 6 complemented by the Figma designs from the designers.

Cypress

As mentioned above, Cypress was the testing framework chosen for the time-travel tests because of its ability to simply set the client (browser) date and time. This enabled the tests to check the ticker, if shown, to be correct immediately before a stage ended and immediately after the next stage started along with other critical elements, the most important of which was the visibility (or not) of the Vote Now button.

The Cucumber "code" for the time travel tests is best explained by looking at the actual code. The code for stages 1 to 3 is shown below. These checks make use of another Cucumber feature: a Scenario Outline. This allows the same logic to be specified (and executed) for each and every row below the Scenario Outline statements. Tags (@r1, @r2, etc) are used to control which stage(s) to run. Typically, when testing a development build, all stages will be executed; when testing a production release then only the current stage tag will be specified on the cypress test runner command line.

Note that the cypress tests were only run on the latest version of Chrome, mostly on Mac OSX Ventura, since we had handed the more comprehensive cross-platform testing to Webdriverio – we'll get to this later.

Below is the Cucumber feature file for the date and time checking for stages 1, 2 and 3. Also checked, since we are in a defined stage when this feature file runs, is whether we can vote or not.

Each Cucumber statement is implemented as a function in Javascript. As an example, the Javscript function that handles the Cucumber statement:

is:

And below is a screenshot captured from a recording of a Cypress run for stage 3, just before voting closes, where the ticker can be seen to be showing 58 seconds to voting closing as set by the Cucumber script above:

The automated tests checked the ticker values were correct using the text that would be read out by a screen reader rather than checking each of the six digits individually. This gave some accessibility coverage to the time-travel tests using the HTML ARIA (Accessible Rich Internet Applications) attributes.

Webdriverio

As mentioned above, Webdriverio was the testing framework chosen to do the page content and visual layout checking. Identical Cucumber scripts with the same Javascript implementations drive both desktop and mobile browsers on multiple operating system versions on MAC OSX and Windows and on real mobile devices (for example iPhone 14, Samsung S22 and other iPad and Android tablets).

The starting reference for the page checking was Confluence and Word documents detailing what components were to be included (and removed) for each of the stages.

Below is an example of the first part of the stage 5 requirements:

And this is how these stage 5 requirements were captured in Cucumber (incomplete):

Like with Cypress, tags were used to divide up the various scenarios so they could be run individually or selectively in a group (@R5 is the tag for stage 5). This is the first part of the scenario that tests voting (this was for production logins – the song choices were not actually submitted for those that are wondering if the tests might have skewed the voting numbers):

The implementation for the Cucumber statement that checks we do not have some text on the page is shown for illustration:

The above code contains a more sophisticated means of handling expect assertion failures, allowing for choosing to stop the scenario on error or continuing after failure. See the next section for more details on why this was added.

It is also worth mentioning that full-page screenshots were able to be captured at run time to aid in detecting visual anomalies, especially for the page-content checks where the page content can look quite different on a mobile device or tablet from that on a desktop browser.

BDD/TDD

A Behaviour Driven Development (BDD) approach was taken with writing the tests with Cucumber, a popular variant of the Gherkin language, being the BDD language of choice. BDD is a development of TDD (Test Driven Development), created by Dan North in 2006. In both BDD and TDD, it is encouraged to write tests before the app features are coded, which usually means all the tests fail in the beginning and as the app is developed more and more tests pass until in the end all tests pass and the app is released to production.

Such an approach was not strictly followed, but most of the date/time and page checking for all six stages was created from the specifications before the app was completed. Taking this approach led to wanting to allow the scenarios to run to completion rather than erroring out at first failure (the alternative would have been to put every check in its own scenario, which was not desirable from a readability or maintenance point of view).

Instead, under the control of a run-time option, scenarios could log a failure to a Slack channel and then continue. This also made it easy to add another check when a bug was discovered that would continue to fail and issue a warning until fixed in the spirit of TDD.

Shown below is a Slack "warning" of a test failing to find the expected text on a page:

A warning shown in a Slack channel

Test Assistant

Over time, the basic test runners provided by both Cypress and Webdriverio have been enhanced to provide the following additional capabilities:

- run the tests on either production or a specific bugfix or feature branch build;
- run on local or cloud-hosted desktop browsers;
- run on cloud-hosted mobile devices (i.e. Android and iPhone devices);
- capture screenshots as the tests run;
- stop on first failure or continue running all tests in keeping with TDD/BDD.

Conclusion

> "If all you have is a hammer, everything looks like a nail"
>
> — The law of the instrument, Abraham Maslow

You have probably heard the idiom above. When initially thinking about how we might test triple j's Hottest 100 of Like A Version, the first step was to identify the capabilities required of the test framework(s) and let this help guide the choice. If automation was/is to be a part of testing – and why would you not choose to do it? -- then these needs will define the testing framework(s) to be used.

In our case, we needed both a hammer and a screwdriver -- both Cypress and Webdriverio were chosen to do the specialised tasks required of the automated testing rather than try and force one testing framework to do both tasks.

Choosing to first write the tests in a high-level language as provided by Gherkin enables the tests to be read, reviewed, maintained and even potentially written by other members of the team, for example, those who write the specifications. This is beneficial with an app such as this (and other members of the Hottest 100 family), as the competitions run only a few times a year. The code is shelved at the end of each competition and then returned to when the development team starts work on the next competition. Gherkin and Cucumber provide a high-level starting point when reviewing what tests were built for the previous competition and what tests are needed for the new competition.

In summary, we used:

- Cypress for its time travel feature;
- Webdriverio for its ability to drive browsers on desktops and real mobile devices using a cloud-based browser provider;
- Gherkin and Cucumber for its ability to capture the tests at a high level and separate what we want to test from how we drive the browsers. Cucumber allows us to switch testing frameworks in the future, should the need arise. The Cucumber statements would remain the same, the implementation would be rewritten to use the replacement test framework's API (application programming interface).

The automated testing techniques described in this article were developed from an earlier set of Cucumber scenarios and associated javascript code created for the Hottest 100 competition run earlier in the year. A large amount of work was saved by re-using nearly all of the javascript functions and most of the Cucumber statements; the biggest change being a consolidation of stages from nine to six.

The automated tests for triple j's Hottest 100 of Like A Version will be reused for future Hottest 100s, ensuring that we can all enjoy these great events from the triple j content and product teams without any technical hiccups.

Posted Tue 27 Jun 2023 at 2:00pm, updated Wed 28 Jun 2023 at 4:10am