Our basic developer implementation guide for client-side A/B tests

This doc assumes you have some basic technical knowledge of how to implement an A/B test including some common coding practices.

Polling

A/B testing 101! It's common in an A/B test to interact with and mutate lots of elements on a page, make sure you are polling for every element before you reference them. It's a good idea to create a separate trigger dependency where all elements are polled first before making changes to the page. This will also ensure a test will "fail gracefully", to protect against future releases where core elements of the page may change.

```
// trigger a poll first to wait for all elements (can be an interval or a mutation observer)
function trigger(cb) {
  poll(['body', 'main > nav', 'main footer.footer li'], cb);
}

// Mutate the DOM when all elements are ready
function initTest () {
  trigger(() => {
    document.querySelector('main > nav').insertAdjacentElement(benefitBar());
    document.querySelector('main footer.footer').insertAdjacentElement(footerNote());
  });
}

initTest();
```

DOM mutations and selectors

- Always be very specific with selectors, and always try to use container selectors when targeting an element. E.g. try to avoid using `.target-element` and consider using `.container-element .target-element` instead.

- If there aren't specific class names, consider using data attributes.

- For modern frameworks such as MUI, it's sometimes common for class names to be auto-generated at build time, this can cause an issue when selecting elements and class names might change in future releases. If this is the case, consider syncing with product dev teams and requesting them to add static

classes to DOM elements specifically for use in experiments (this is a also common practice for e2e testing tools such as Cypress)

Mutation observers

- Mutation observers are used to apply a desired effect after the DOM has changed, common scenarios include applying a change after a page re-renders or applying code later to a page after a certain element renders into view.
    - It's generally best to avoid using mutation observers where possible, consider using alternative methods such as AJAX complete if the situation allows for it.
    - Ensure your observer callback is optimised for efficiency and triggers the minimum number of times necessary. This can be done by adding in extra checks within the callback itself or passing a config as shown below to minimise the number of executions.
    - Always add safety checks to your callbacks, it's always a good idea to run a simple URL path check before you run your change, this is because observers can continue to fire if left unchecked, especially on SPAs where the observer may stay connected across pages:

```
const callback = (mutationsList, observer) => {

  // Add a safety check here before executing your callback change
  if (!location.pathname.includes('/desired-path')) return;

  for (const mutation of mutationsList) {
    if (mutation.type == 'childList') {
      fn(observer);
    }
  }
};

const observer = new MutationObserver(callback);

// It's a good idea to pass in config to make sure your callback does not fire
// too many times, make sure to test beforehand
observer.observe(targetNode, {
  attributes: true,
  childList: false,
  subtree: false,
});
```

Running tests across pages

- It's common for A/B tests to run across multiple pages where custom configuration is needed per page. Test logic can get convoluted in these scenarios, for complex tests, it's always a good idea to have separate polling conditions per page and have page logic self- contained to ensure code is readable and clean:

```
const pageTypes = [
  {
    path: '/quote/policy-details',
    pageType: 'POLICY_DETAILS',
    pollElements: [selectors.nav, selectors.policyMain]
  },
  {
    path: '/quote/personal-details',
    pageType: 'PERSONAL_DETAILS',
    pollElements: [selectors.nav, selectors.cards],
  },
];

function getPageType() {
  const locationPath = window.location.pathname;
  return pageTypes.find(page => page.path === locationPath) || {};
}

// Simple business logic makes it readable to see what changes are being applied!
function start(pageType) {
  if (pageType === 'POLICY_DETAILS') {
    policyDetails();
  } else if (pageType === 'PERSONAL_DETAILS') {
    personalDetails();
  }
}

// polling conditions
function trigger() {
  const { pollElements, pageType} = getPageType();
  if (pollElements) {
    poll(pollElements).then(() => {
      start(pageType);
    });
  }
}

trigger();
```

Tests depending on other winning tests

- It's a **bad idea** to have an A/B test depending on another live test. Suppose you have a winning test running at 100% that adds a new sitewide navigation and you want to run an iteration on the newly created navigation in a future test. In this scenario, it's easy to build a new test that references the new navigation, but that will create a dependency on the winning test. If the winning test is paused or modified then your new test is also vulnerable to breaking. This is a relatively simple example, but this can get complex with lots of tests depending on each other and also creates problems with test execution order. In our navigation iteration scenario, the best thing to do is as follows:
    - Pause the 100% winning test
    - Copy the code from the 100% winning test into your new test so that the code runs on control and variation. In most platforms there is an option to run control code, in others, you may need to physically create a new variant named "control" .
    - Rather than running your new iteration changes on top of the previous code, it's best to directly modify the code itself in the variation to prevent execution order problems.

  This method keeps the execution order clean and removes test dependency.

Continuous development integration for client-side tests

Deploying A/B tests to a platform does not normally require a standard CI/CD automation process as the test code lives separately from the product code in an A/B testing platform. That being said we highly recommend implementing the following practices:

- Source control your tests and push them to a repo that acts as a reference for all A/B tests. This makes it very easy to update existing tests and keep track of code.

- We do not recommend coding A/B tests directly into the platform as part of the build process. We recommend using your code editor with a transpiler like Webpack or Rollup so you can make use of modern JavaScript. Code should then be bundled into the platform before launch which should also account for browser coverage.

- We recommend having a main branch that acts as the single source of truth for experiments; new tests in development should have their own branch before merging into the main branch.

- We recommend having code peer-reviewed just like any other standard development process.

Quality Assurance (QA)

Quality Assurance (QA) in the context of AB experimentation is a critical process that ensures the reliability, accuracy, and overall effectiveness of the tests before they are deployed to your customers. QA involves a meticulous examination of various elements within the AB test. By rigorously testing each variant and validating the experiment's functionality, you will aim to identify and address any potential issues or anomalies that could compromise the integrity of the test results. This proactive approach not only safeguards against errors but also enhances confidence in the outcomes, ultimately allowing you to make informed decisions based on reliable data.

Below are some top tips when going through the QA process.

Test Scripts & Scenarios

One crucial aspect of the QA process involves the creation and execution of comprehensive tests scripts. Test scripts serve as detailed guidelines for systematically assessing the functionality and performance of each variant in an AB test. These scripts outline specific scenarios and user interactions that need to be tested, ensuring a thorough examination of the experiment under various conditions. By crafting test scripts, you can simulate user journeys allowing you to identify potential issues before the test goes live.

This process is particularly important in scenarios where changes in functionality may impact the results of the experiment. Through rigorous testing of these scripts, you can uncover any unintended consequences and address them proactively.

Metrics

Before initiating the QA process, it is imperative to review the metrics and set up of the AB test within the platform. This pre-emptive check serves a critical role to ensure that the experiment aligns with its intended goal and accurately captures the relevant metrics for analysis. This check involves validation of the tracking mechanisms, conversion events and KPIs defined for the test. Verifying the correctness of the metric implementation helps to avoid skewed or inaccurate data, ensuring the reliability of the insights from the experiment.

Emulation vs. Real Devices

Emulated devices offer the advantage of cost effectiveness and the ability to simulate a wide range of device configurations, enabling you to cover a broader spectrum of potential user scenarios, along with the ability to automate the QA process. However, emulators may not fully replicate the nuances of real world device behaviour, leading to potential blind spots in testing. Striking a balance between emulated and real device testing is crucial to achieving comprehensive QA coverage, reducing the likelihood of overlooking issues that might only appear on real devices.

Live QA

Once the AB test is live, completing a thorough QA run through becomes an integral part of the QA process. This final stage is designed to validate that all components continue to function as intended.

Protecting A/B tests against releases

- Before an A/B test goes live to production, make sure to target the test to lower environments too (for example UAT or STAGING), this is to make sure the A/B test is fully QA'd with up-and-coming releases. QA testers should always test releases against live A/B tests and they should be able to force bucket into a variation within the test environment.

- Ensure that all live tests are documented internally and that other product teams are aware of what is going live.

- Implement fail-safe strategies into your test code; when live tests are running in the platform it's a common problem where a test may cause a JavaScript error due to a particular DOM element being removed in a release. Ensure that a test "fails gracefully" , by making use of try-catch blocks and implementing a polling trigger that polls for all DOM elements being referenced.

Pre-launch checklist

- ☐ Fail-safe code - have you implemented polling to check for the presence of all DOM elements that you are modifying in the test?
- ☐ Has the developer done their own sanity check across a few different browsers and devices?
- ☐ Has your code been reviewed by another developer?
- ☐ Has the test been QA'd by a dedicated tester?
- ☐ Has the test been exposed to lower test environments to protect against future releases?
- ☐ If you have a dedicated testing team, are they aware of the test about to be deployed?
- ☐ Have you documented that your test is going live so other teams are aware?
- ☐ Is your test "self-contained" and not depending on other winning tests that are live in the testing platform?