

Building an experimentation framework for web apps - Tutorial Notes

OSCON 2011

Zhi-Da Zhong, zz@etsy.com

Why?

Questions we want to answer

- “What will happen if I do X”? Will the stie crash? How will the users react?
- “Is X better than Y?” Better could mean more clicks, more \$, retention, etc. - depends on what your goal is.

We're bad at predicting the future and knowing the consequences of alternative actions.

Experiments

- Since we don't know, we set up experiments so we can try things out and gather real data, i.e., the reactions of real users, instead of speculating on what will happen or would've happened.
- Specifically, we want to run experiments where we present different alternatives to different people. Why different? Because otherwise we can't establish correlations between what we present and their reactions.

What can we test in web apps?

Front end experiments

- These are most common. They include changes in layout, colors, images, etc.
- There are typically no functional changes associated with them.
- Pure design changes can have surprisingly high impact. (And the winner can be surprising: e.g., a plain-looking design can win over much fancier ones.)

Somewhat more complex tests

- Tests that involve multiple pages.
- Functionality might be different.

Backend experiments

- Backend changes can (obviously) affect the site as well. So why not deploy them as experiments, too?
- We can change algorithms, architectures, batch processes, etc.

Example: Etsy search backend

A complete rewrite involving * new algorithm that looks at a different set of fields * new RPC protocol

(Thrift) between web app and backend search servers * changes in what the search servers return (different fields) * upgraded to a new snapshot of Solr trunk

We ramped the new cluster up and down a few times until we got it right.

DB re-architecture

We've also been migrating from a single monolithic Postgres DB to sharded MySQL servers.

- We tee the writes and read from either the old or the new DBs.
- We do it for one subset of the data at a time. Sometimes by user, sometimes by other entities.
- There are multiple experiments/ramp-ups involved for different parts of the DB.

New features

We can test entire new features involving frontend and backend work. We make the feature available to a subset of users.

Not just 2 variants

- We can test any number of variants: A/B/C...
- Multi-variate tests are a special case of tests having >2 variants. We want to experiment with multiple aspects of e.g., a page at the same time, and find the optimal combination of values. The key to make the length of the test manageable is to pick the combinations based on our knowledge of which variables are independent.

Caveats

- Sometimes the content you want to test isn't under your control. E.g., user-contributed content. We can't just alter them.
- Price tests are tricky. People might get upset and demand refunds.
- Some things are hard to measure/quantity: e.g., happiness.
- Sometimes we're interested in long-term impact - beyond the typical length of an A/B experiment.

Other ways to test

- We can enable the feature for internal staff only.
- Similarly, we can make it available to a few whitelisted users - internal or external.

Opt-in experiments

- We can also let users opt in to an experiment.

Other complementary techniques

- Instead of showing different people different things, we can show a group of people the same thing, and observe their reactions. This can be done online.
- We can also show 2 alternatives to the same person, and ask them to compare them. E.g., our side-by-side tool for search.

These different experimentation methods help us learn different things.

How to implement A/B testing

A common approach

- Drop some JS snippet onto the page you want to test. And it does the rest. Backend changes aren't necessary (or easy).
- Typically there's a (web-based) UI for non-technical users.
- Often aims to minimize involvement of developers.

This approach can be good for simple design tests. But it's limited in scope and trying to take the developer out of the process doesn't really make a lot of sense.

Our approach

- Some developer is already involved in building the feature to be tested. So let's just make it easy for that developer to set up the test. It's also a developer tool, not just a marketing tool.
- Instead of special configuration data and UI, we make configuration part of the code. So no XML parsers, etc. to depend on. No special UI to build. Code is more expressive than typical config data. We can also take advantage of the versioning system that's in place for code: from the version number, we can find out the exact combination of code + config at any given time. We get history of the tests for free.
- Experimentation has become part of our normal development/deployment process. Every change can be an experiment.

What it looks like

It's a process of going from (old) default => experiment => (new) default.

To add a new feature

We start by adding a 'feature flag' to the config file. Then we add a simple if statement to the code where we want to introduce the feature/change. We branch in code.

Then we deploy the new feature flag. Nothing should change, behavior-wise.

Now that we have the flag in place - and turned off for the moment, we can implement our new feature/change - and deploy code that's still in development without worrying about it affecting users. (But this doesn't mean we should be completely reckless - don't destroy the network or the DB, for example.)

Internal testing

We make a simple configuration change to make the feature available to internal users. Note there's no code change in the app.

Similarly, we can add a whitelist to give specific users access. Again, just a config change.

Opt-in experiments

Sometimes we expose the feature as a public opt-in experiment that users can sign up for. Also another line of config.

A/B tests

Then finally we can run a real A/B test...by adding another config line. Percentages are specified with sub-1% precision, so we can start with a really small group for risky, high-traffic features.

Then, if we find that it works, we turn it on for everyone.

If multiple ways of testing are specified, a specific order is followed, we first check whitelist (or blacklist), internal status, then opt-in status, and finally the randomized selection for A/B tests.

The framework

The overall structure is quite simple:

1. Select some variant for each user/subject.
2. The app acts according to the selection.
3. Log the event so we can analyze the data.

A test is represented by a set of *variants*, a way of identifying the subjects, a way of *selecting* the variants for each subject, and a logger.

Variants

- A variant contains a set of (arbitrary) key-value pairs representing properties that are interpreted by the app. E.g., number of columns on a page.
- It has a name so it can be logged.

SubjectIdProvider

A simple abstraction that lets you identify test 'subjects':

- they're often users, but not always. (E.g., images.)
- there might be different types of users that have different roles in an app and might even have different ID spaces.
- there are also different ways of identifying a user, e.g., user ID if they're registered and signed in, or some cookie value if they're not.

We use these IDs for logging as well as (sometimes) deciding which variant each subject should get.

Selectors

It's a very simple interface with 1 function that simply maps a subject ID to a variant name.

Combining selectors

As we've seen, we can specify multiple ways of selecting variants at a given time. So how do we combine them?

- OR is obvious...and almost works. Except blacklists wouldn't work.
- AND has the opposite problem: it breaks whitelists.
- So we combine them as a sequence: we try each one in the defined order, and returns the first non-null selection. Each selector in the sequence either returns something or null if it decides it's not applicable to that particular subject.

Loggers

We record the triplet (test, variant, subject).

We want to be able to log in different ways:

Access logs

- We insert test-variant info into the Apache access log using the 'note' facility.
- There are lots of ways to analyze access logs, from grep to fancy commercial tools.

3rd-party analytics integration

- Integrating with 3rd-party analytic tools (that you might already be using) can be a quick way to start doing A/B testing.
- It may be cheap/free to use.
- But it may or may not be able to handle the volume you have, offer the lag time you desire, or lets you customize it to the extent you want.

How to integrate with 3rd-party analytics

- You can usually define some custom variables where you can put the test and variant names. But take note of restrictions on the number of variables you can have and the sizes of variable names/values.
- You can then segment your users based on the custom A/B variable(s).
- There are built-in metrics (e.g., click rate, abandonment %, etc.) you can look at.

Google analytics example: * some boilerplate JS set up * a single line to add a custom variable containing A/B data.

Etsy's custom event tracking framework

- Our pages containing JS that sends event beacons to the web server.
- The event log is analyzed in the cloud using Hadoop.
- We pull the results down and create custom reports/dashboards.

Building on top of the core API

Some additional things you might want to build to make setting up tests quick and easy.

Test builders

- Common usage patterns

The core API is designed for generality: you can have different ways of identifying users, selecting variants, and logging to different destinations. Typically, though, for each app/site, a small set of implementations are used in most tests. E.g., you might have some ID provider for signed-in users, and another for non-signed-in ones. Or even just a single cookie-based ID provider for all visitors. Test builders let you capture and reuse these common implementations. And they give you a simpler, higher-level API. E.g., you can have a test builder for simple A-B (i.e., 2-variant) tests, and all you need to do in the app is something like our feature flag call `$cfg->isEnabled('test.name')`.

- Test design

Beyond encapsulating Test instantiation boilerplate, they can also perform more sophisticated test generation tasks.

Weight equalization - Some people like to have equal-weighted variants. E.g., (A=10%, B=90%) because (A=10%, B1=10%, B2=80%). (It can make analysis a bit easier.) We can write a tool to split variants as necessary to equalize the weights.

Multivariate tests - We can write a test generator that takes the list of variable to be tested, their values and their dependency matrix as input, and create a set of variants that represent combinations of those variables. (We haven't had a need for this type of generator yet.)

Automatic dispatchers

As an alternative to putting if blocks in the code, we can write a generic dispatcher that intercepts a call, calls the core A/B API to obtain the variant, and then delegates to the appropriate component that implements the functionality of that variant.

Example: in apps using MVC frameworks like Spring, you can implement a custom URL dispatcher that invokes the appropriate controller or view, depending on the variant that's chosen for a particular request. The controller/view name can be specified as a variant property.

A dispatcher works with a particular component interface, and so it's less flexible in terms of granularity.

It can make tests seem like magic because the developer of the individual components don't need to do - or even know - anything special about the test setup.

Selector registry

It can be helpful to have a place where all the available selectors are listed. It helps to prevent re-inventing the wheel. It also serves as a kind of documentation, and tells the developer where to look for the implementation.

Randomized Selector

What does 'random' mean in a randomized controlled experiment?

- The choice of variant is independent of subject attributes. E.g., the subject's gender, location, etc. should not bias the selection.
- There shouldn't be any correlation between the variants chosen for one test and those of another. In other words, the likelihood that a subject lands in any particular variant for one test should be the same regardless of what variants that subject gets in any other tests.
- The result of variant selection shouldn't depend on what hour/day/month/... it is. (But it's probably ok to depend on say, the nanosecond field of the time of a web request.)

Persistence

But simply picking variants randomly isn't enough. If someone comes to the site and see one design, generally we want to show that visitor the same thing during their next visit.

- Inconsistency is generally a bad experience.
- The data will also be less useful. If someone sees multiple variants of a test, then we'll have trouble attributing their actions to a specific one.
- There are also tests that may involve different parts of a site that are visited over multiple page views or sessions. We obviously want them to be consistent.
- But sometimes we *have to* break persistence. E.g., if something isn't working and we decide to turn it off.

Ramping up/down

When we start testing a feature/change, we generally start with a small %, and then depending on how well things are working, adjust the % up or down. Sometimes we only discover a problem (e.g., drop in performance) after we increase the %, and we'll have to ramp it back down.

- This allows us to reduce potential damage of a risky change.
- We can also use this to spread load over time, e.g., if the change involves some one-time, relatively expensive data migration task.

Persistence + Ramping

When we change the relative weights of the variants, we'll generally have to break persistence and move some users from their one bucket to another. (In some case, it may be possible to achieve the desired change in weights by changing the weights only for new visitors, but it can take a long time to do that.)

Ideally, when we change the %, we should try to re-assign only as many users as necessary to minimize overall inconsistency:

- If we're ramping up a feature, for example, we should just add users to that feature, and not take it away from anyone who already has it.
- If we're ramping down, then the opposite: just remove it from a subset of the users who currently have it.

rand()

If we do the obvious thing and use rand() to make our random selections, we'll have to explicitly persist that choice for each user/subject. We can save that choice in a cookie and/or the DB, for example. But this approach has some problems:

- As the number of tests grows over time, we don't want the number/size of cookies to grow indefinitely. A similar problem exists for DB. So we'll have to perform maintenance to keep the sizes in check.
- Ramping up/down can also be tricky with this approach, esp. if the variant is saved in a client-side cookie.

Hashing

Hashing is another way to achieve randomization. The basic idea is simple, deterministically compute the variant from the subject ID.

- Obviously, we automatically get persistence for each subject (ID).
- We also get subject attribute independence. But what if the subject ID isn't random? What if there's a correlation between it and some subject attribute? The answer is to use a good hash function that maps small changes in the ID input to large changes in the output.
- What about test independence? We now add the test ID to the input so the output is different for different tests.
- We also need to take weights into account. We need the output to be distributed according to the desired weights of the variants. The answer is to add a second step, partition after hashing.

Hash + Partition details

1. Hash the input (test ID + subject ID) into the interval $[0, 1)$.
2. Partition that interval according to the weights.
3. Assign each subject to the variant that corresponds to the partition that the hash falls in.

Ramping

If we change the weights, we change the sizes of the partitions, and consequently the hash values that each one captures.

Which hash function

- Lots of possibilities, including all the usual crypto hash functions.
- Test the hash function by computing the variants for a given set of tests and subject IDs. Check the output distribution and look for correlations between tests.
- But be careful as the results can depend on the test data.

We picked SHA-256.

A/B + opt-in

Doing both an opt-in experiment and a regular A/B test presents a problem: we need to separate the people who opted-in from those who got random assignments so that self-selection doesn't skew the results.

Our solution is to create different sets of variants for the opt-in experiment and the A/B.

- Logging the different variant names allows us to separate the users.
- The application code will check the values of variant *properties* rather than their names so the app doesn't need to distinguish opt-ins and regular A/Bs.

(This is actually a problem with internal and whitelisted users as well, but their numbers are generally small compared to 'real' users so the impact is less severe.)

Analysis

2 basic things to know about (or to know to ask your neighborhood mathematician about):

Confidence Intervals

We need to calculate not just the value of whatever metric we're interested in, but very importantly, some measure of how certain we're about the result. Confidence intervals are a way to measure that. It tells us what range of values we can expect if things were just happening randomly. E.g., if some metric is 1.5, it can tell us there's a 95% chance that any observed value between 1.4 and 1.7 can be due to random fluctuations, and not any real difference between different variants of the feature in question.

Binomial experiments

They're like coin tosses. Each trial is independent from any other. In the web context, they might be, e.g., the initial visit from different users. They're relatively easy to analyze, and cover a lot of scenarios.

Test design tips

What's the question?

- First we need to be sure we know what question we're trying to answer. Conversion rate? Time on site?
- And how much difference in the metric are we trying to detect? Are we trying to find out if X is 5% better than Y?

Who?

Different groups of users may react differently to the same functionality.

- They may have different roles on the site. E.g., buyers vs sellers.
- Old and new users may react differently due to habit, expectation, and novelty factors.

Whose responses are we trying to determine?

When?

We may get different results at different times:

- Different types of users may visit the sites.
- The same user may behave differently, e.g. during week day lunch hour and weekends.
- The site content may vary at different times.
- Site performance might vary to the point that it influences user behavior.

Running a test for one or more full weeks is often a good idea in many cases.

Summary

Experimentation allows us to

- take more risks and be more innovative
- improve mean-time-to-recovery by helping us pinpoint the source of a problem
- build and deploy changes with less fear & stress

And we can build a small, relatively simple framework to do that.

