PRIMER

Basics of Feature Flags & Feature Toggles

At the core of Split is a relatively new engineering practice known by many names: 'feature toggles', 'feature switches', 'feature flippers'. To make it simple, from here on out we'll call them one thing: feature flags.

Defining 'Feature Flags'

Feature flags (aka toggles, flips, gates, or switches) are a software delivery concept that separates feature release from code deployment. In plain terms, it's a way to deploy a piece of code in production while restricting access—through configuration—to only a subset of users. They offer a powerful way to turn code ideas into immediate outcomes, without breaking anything in the meantime.

History

Historically, software was developed on 'long lived' feature branches. The branch was long lived because it existed for the entire lifecycle of the feature's development, from first commit to feature completion. It wasn't merged into master until QA had validated it in an integration environment.

Feature branches offered isolation of features, however they were a nightmare to manage, merge, and test. Imagine multiple teams trying to merge their branches into master at the same time. It delayed integration, created contention for QA resources and slowed down product development.

Feature flags were created to address these pains. They offered a way to continuously merge a feature into master, deploying it to production while keeping it 'dark', and then turning it on only when the feature was ready for customers.

How long has the idea been around? One of the earliest mentions of feature flags is an article from Flickr back from 2009, and the author and speaker Martin Fowler wrote about them back in 2010.

Code Example

Say an e-commerce company is rolling out a new algorithm to recommend products to customers. Here is a quick example of how they could use a flag to control access to this feature:

```
if flags.isOn('user-id', 'new-recommendation-algorithm') {
  return new_recommendation_algorithm('user-id');
} else {
  return old_recommendation_algorithm('user-id');
}
```

Here 'flags' is an instance of a class that evaluates whether a user has access to a particular feature or not. The class can be configured by something as simple as a file or database backed configuration to a distributed system with a UI.

Similar examples can be imagined for other languages.

Use Cases for Developers

At a high level, feature flags accelerate development by powering the following use cases:

Trunk / Mainline Development

All development happens on a single shared branch (trunk, master, head etc.) with feature branches being created only for pull requests. Trunk development helps teams identify problems early and hence, move faster. Feature flags are a requirement for trunk development.

Continuous Delivery

Continuous Delivery is the concept that every push to master is built and deployed to production. This can only happen if unfinished features are behind flags that are turned off by default, otherwise continuous delivery can turn into continuous failure.

Testing in Production

New feature releases are preceded by functional QA and performance testing. This testing requires the creation of a UAT or staging environment with a sample of production data. This sampling and copying of production data is problematic: first it's a red flag for data privacy and security conscious teams. Second, these staging environments are not a faithful replica of production infrastructure, so any performance evaluations have to be taken with a grain of salt.

Feature flags allow teams to perform functional and performance tests directly on production with a subset of customers. This is a secure and performant way of understanding how a new feature will scale with customers.

Kill a Feature

By having a feature behind a flag, it can not only be rolled out to subsets of customers, it can also be removed from all customers if it is causing problems to customer experience. This idea of 'killing' a feature is better than having to do an emergency fix or a code rollback.

Use Cases for Product Managers

For Product Managers, feature flags enable the following powerful use cases.

Experimentation

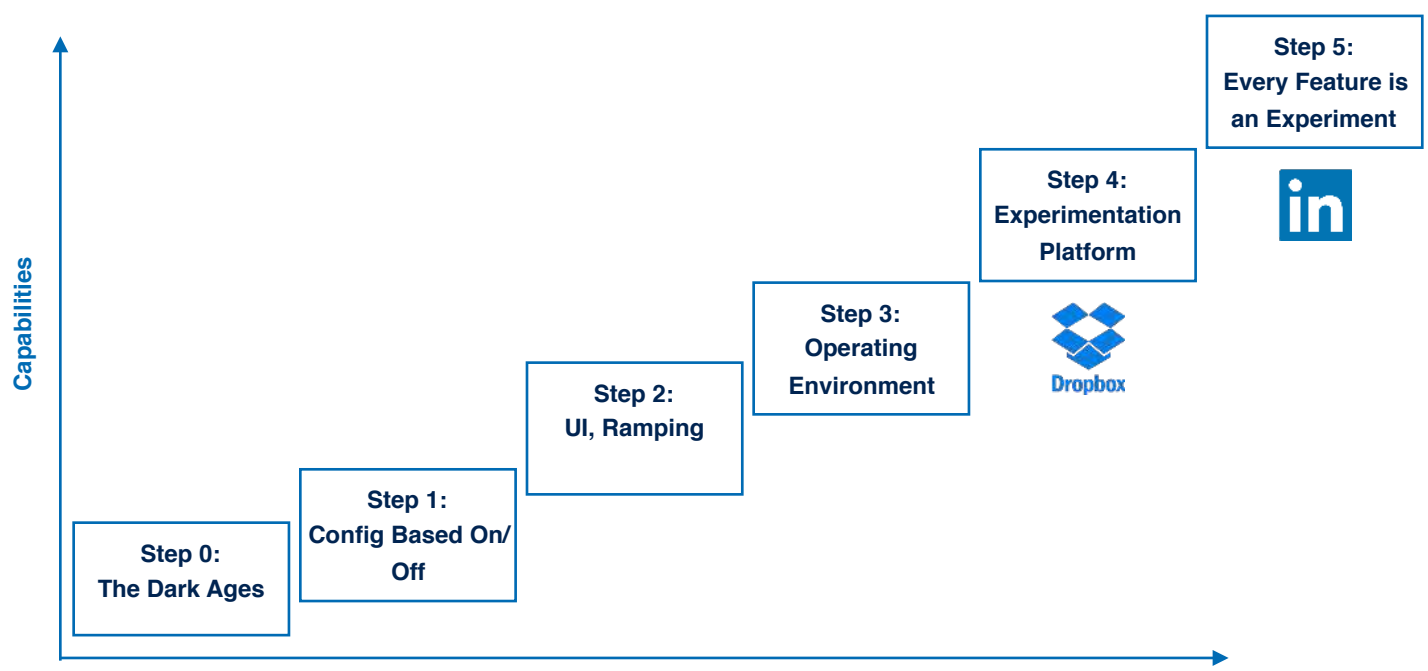
By creating a subset of customers that are exposed to a feature and a subset that is not, feature flags become the basis of full-stack experimentation. The impact of, say, a new recommendation system, can be measured on customer experience metrics: latency, error rates, conversions, engagement, growth, subscription rates, and revenue. The outcome of any idea can thus be measured.

Paywalls

Feature flags can be used to permanently tie features to subscription types. For instance, a feature could be available to every customer as part of a 30 day free trial, but is gated afterwards by the customer buying a premium subscription.

Evolution of Feature Flagging Systems

Most teams invest in building their own feature flagging systems. This image shows the evolution of these systems from flagging to controlled rollout to experimentation platforms.



Step 1: Configuration-based On/Off Flags

As step 1, most teams build a basic on/off flag system that is configured outside of code. This configuration is often done via a file or a database. A feature is either on for all customers or off for them. To make a change, an engineer or a DBA is needed to turn a feature on or off, which makes this system inaccessible to non-technical teammates like PMs.

Step 2: Basic UI & Percentage Targeting

Eventually, engineers get tired of fielding requests from PMs to enable/disable features. Plus, making changes in a file or database can be error prone. So, teams build a basic UI that allows PMs to turn a feature on/off without engineering support. Investments in targeting capabilities of the flag allow teams to turn a feature on just for internal employees, specific beta users, or percentages of customers.

Evolution of Feature Flagging Systems (Cont.)

Step 3: Permissions & Tagging (aka a 'Controlled Rollout Platform')

As usage of the feature flagging system grows, errors become common. An engineer accidentally kills the features of another team. A PM accidentally rolls out a feature to all customers when they only wanted to release it to a subset. The team realizes that they need to treat the flagging system as a production system: with permissions, tagging, and integrations with team paging systems (like Slack and PagerDuty).

Step 4: Evolution of Feature Flagging Systems

Eventually, teams realize that feature flags can serve their experimentation needs. The targeting system is expanded to support arbitrary treatments ('a' or 'b' or 'c') instead of a plain on/off flag. Basic analytics are added to measure the impact of an experiment (another term for a flag) on a a few primary metrics driving the company (e.g. conversions, growth, revenue etc.). Targeting capabilities are enhanced to allow defining of customer cohorts by any customer dimension. At this point, the platform is valuable to engineers, PMs, and SREs alike. Good industry examples are Dropbox's Stormcrow and Airbnb's Trebuchet.

Step 5: Every Release is an Experiment

The last step in this evolution is when the entire company views every feature release as an experiment. The impact of every release is measured on a cross-functional suite of metrics, from engineering to product to business metrics. PMs not only watch movement on the metric they are focused on driving, but also watch out for potential negative impact on metrics important to the rest of the company. Experiment design and statistical significance becomes an important aspect of the system. Great industry examples are Facebook's Gatekeeper and LinkedIn's LiX.

Implementation Notes

If you are building your own feature flagging system, it is important to be take these practical things into consideration.

Evolution of Feature Flagging System

If you keep the evolution of these systems in mind, you will be able to design a better system today that will continue to evolve with your team's needs.

B2B vs B2C companies

The needs of B2B companies are a bit more complicated than those of B2C companies. They may need to rollout features user-by-user, or account-by-account. The latter case is unique in that if a feature is rolled out to an account, it should be accessible to all users within that account. If a user exists in multiple accounts, you have to think about the impact on user experience as the user switches from account to account.

Stickiness

It's important that your system supports stickiness: if a user is shown a feature, they should continue seeing the feature no matter which machine their request is routed to. Without stickiness, the user experience can be very inconsistent.

Remember to Remove Flags

If flags are left in the code for too long, they lead to tech debt. Old branches of code may no longer work and if someone accidentally kills a feature, the customer experience will be negatively impacted. If you are building a solution for your team, think about building a notification system that helps teams do house cleaning.

About Split

Split is the leading platform for feature experimentation, empowering businesses of all sizes make smarter product decisions. Companies like Vevo, Twilio, and LendingTree rely on Split to securely release new features, target them to customers, and measure the impact of features on their customer experience metrics. Founded in 2015, Split's team comes from some of the most innovative enterprises in Silicon Valley, including Google, LinkedIn, Salesforce and Databricks. Split is based in Redwood City, California and backed by Accel Partners and Lightspeed Venture Partners. To learn more about Split, contact hello@split.io, or get started for free at www.split.io/signup.