

---

Feature is no longer actively maintained and is no longer in sync with the version used internally at Etsy.

## Feature API

Etsy's Feature flagging API used for operational rampups and A/B testing.

The Feature API is how we selectively enable and disable features at a very fine grain as well as enabling features for a percentage of users for operational ramp-ups and for A/B tests. A feature can be completely enabled, completely disabled, or something in between and can comprise a number of related variants.

For features that are not completely enabled or disabled, we log every time we check whether a feature is enabled and include the result, including what variant was selected, in the events we fire.

The two main API entry points are:

```
1 Feature::isEnabled('my_feature')
```

which returns true when `my_feature` is enabled and, for multi-variant features:

```
1 Feature::variant('my_feature')
```

which returns the name of the particular variant which should be used.

The single argument to each of these methods is the name of the feature to test.

A typical use of `Feature::isEnabled` for a single-variant feature would look something like this:

```
1 if (Feature::isEnabled('my_feature')) {
2     // do stuff
3 }
```

For a multi-variant feature, within the block guarded by the `Feature::isEnabled` check, we can determine the appropriate code to run for each variant with something like this:

```
1 if (Feature::isEnabled('my_feature')) {
2
3     switch (Feature::variant('my_feature')) {
4         case 'foo':
5             // do stuff appropriate for the foo variant
6             break;
7         case 'bar':
8             // do stuff appropriate for the bar variant
9             break;
10    }
```

---

```
11 }
```

It is an error (and will be logged as such) to ask for the variant of a feature that is not enabled. So the calls to variant should always be guarded by an `Feature::isEnabled` check.

The API also provides two other pairs of methods that will be used much less frequently:

```
1 Feature::isEnabledFor('my_feature', $user)
2
3 Feature::variantFor('my_feature', $user)
```

and

```
1 Feature::isEnabledBucketingBy('my_feature', $bucketingID)
2
3 Feature::variantBucketingBy('my_feature', $bucketingID)
```

These methods exist only to support a couple very specific use-cases: when we want to enable or disable a feature based not on the user making the request but on some other user or when we want to bucket a percentage of executions based on something entirely other than a user.) The canonical case for the former, at Etsy, is if we wanted to change something about how we deal with listings and instead of enabling the feature for only some users but for all listings those users see, but instead we want to enable it for all users but for only some of the listings. Then we could use `isEnabledFor` and `variantFor` and pass in the user object representing the owner of the listing. That would also allow us to enable the feature for specific listing owners. The `bucketingBy` methods serve a similar purpose except when there either is no relevant user or where we don't want to always put the same user in the same bucket. Thus if we wanted to enable a certain feature for 10% of all listings displayed, independent of both the user making the request and the user who owned the listing, we could use `isEnabledBucketingBy` with the listing id as the bucketing ID.

In general it is much more likely you want to use the plain old `isEnabled` and `variant` methods.

For Smarty templates, where static methods can't readily be called, there is an object, `$feature`, wired up in `Tpl.php` that exposes the same four methods as the Feature API but as instance methods, for instance:

```
1 {% if $feature->isEnabled("my_feature") %}
```

## Configuration cookbook

There are a number of common configurations so before I explain the complete syntax of the feature configuration stanzas, here are some of the more common cases along with the most concise way to write the configuration.

---

**A totally enabled feature:**

```
1 $server_config['foo'] = 'on';
```

**A totally disabled feature:**

```
1 $server_config['foo'] = 'off';
```

**Feature with winning variant turned on for everyone**

```
1 $server_config['foo'] = 'blue_background';
```

**Feature enabled only for admins:**

```
1 $server_config['foo'] = array('admin' => 'on');
```

**Single-variant feature ramped up to 1% of users.**

```
1 $server_config['foo'] = array('enabled' => 1);
```

**Multi-variant feature ramped up to 1% of users for each variant.**

```
1 $server_config['foo'] = array(  
2     'enabled' => array(  
3         'blue_background' => 1,  
4         'orange_background' => 1,  
5         'pink_background' => 1,  
6     ),  
7 );
```

**Enabled for a single specific user.**

```
1 $server_config['foo'] = array('users' => 'fred');
```

---

**Enabled for a few specific users.**

```
1 $server_config['foo'] = array(  
2     'users' => array('fred', 'barney', 'wilma', 'betty'),  
3 );
```

**Enabled for a specific group**

```
1 $server_config['foo'] = array('groups' => 1234);
```

**Enabled for 10% of regular users and all admin.**

```
1 $server_config['foo'] = array(  
2     'enabled' => 10,  
3     'admin' => 'on',  
4 );
```

**Feature ramped up to 1% of requests, bucketing at random rather than by user**

```
1 $server_config['foo'] = array(  
2     'enabled' => 1,  
3     'bucketing' => 'random',  
4 );
```

**Single-variant feature in 50/50 A/B test**

```
1 $server_config['foo'] = array('enabled' => 50);
```

**Multi-variant feature in A/B test with 20% of users seeing each variant (and 40% left in control group).**

```
1 $server_config['foo'] = array(  
2     'enabled' => array(  
3         'blue_background' => 20,  
4         'orange_background' => 20,  
5         'pink_background' => 20,  
6     ),  
7 );
```

---

### New feature intended only to be enabled by adding ?features=foo to a URL

```
1 $server_config['foo'] = array('enabled' => 0);
```

This is kind of a funny edge case. It could also be written:

```
1 $server_config['foo'] = array();
```

since a missing 'enabled' is defaulted to 0.

### Configuration details

Each feature's config stanza controls when the feature is enabled and what variant should be used when it is.

Leaving aside a few shorthands that will be explained in a moment, the value of a feature config stanza is an array with a number of special keys, the most important of which is 'enabled'.

In its full form, the value of the 'enabled' property is either the string 'off', meaning the feature is entirely disabled, any other string, meaning the named variant is enabled for all requests, or an array whose keys are names of variants and whose values are the percentage of requests that should see each variant.

As a shorthand to support the common case of a feature with only one variant, 'enabled' can also be specified as a percentage from 0 to 100 which is equivalent to specifying an array with the variant name 'on' and the given percentage.

The next four most important properties of a feature config stanza specify a particular variant that special classes of users should see: 'admin', 'internal', 'users', and 'groups'.

The 'admin' and 'internal' properties, if present, should name a variant that should be shown for all admin users or all internal requests. For single-variant features this name will almost always be 'on'. (Technically you could also specify 'off' to turn off a feature for admin users or internal requests that would be otherwise enabled. But that would be weird.) For multi-variant features it can be any of the variants mentioned in the 'enabled' array.

The 'users' and 'groups' variants provide a mapping from variant names to lists of users or numeric group ids. In the fully specified case, the value will be an array whose keys are the names of variants and whose values are lists of user names or group ids, as appropriate. As a shorthand, if the list of user names or group ids is a single element it can be specified with just the name or id. And as a further shorthand, in the configuration of a single-variant feature, the value of the 'users' or 'groups' property can simply be the value that should be assigned to the 'on' variant. So using both shorthands, these are equivalent:

---

```
1 $server_config['foo'] => array('users' => array('on' => array('fred')));
```

and:

```
1 $server_config['foo'] => array('users' => 'fred');
```

None of these four properties have any effect if `'enabled'` is a string since in those cases the feature is considered either entirely enabled or disabled. They can, however, enable a variant of a feature if no `'enabled'` value is provided or if the variant's percentage is 0.

On the other hand, when an array `'enabled'` value is specified, as an aid to detecting typos, the variant names used in the `'admin'`, `'internal'`, `'users'`, and `'groups'` properties must also be keys in the `'enabled'` array. So if any variants are specified via `'enabled'`, they should all be, even if their percentage is set to 0.

The two remaining feature config properties are `'bucketing'` and `'public_url_override'`. Bucketing specifies how users are bucketed when a feature is enabled for only a percentage of users. The default value, `'uaid'`, causes bucketing via the UAID cookie which means a user will be in the same bucket regardless of whether they are signed in or not.

The bucketing value `'user'`, causes bucketing to be based on the signed-in user id. Currently we fall back to bucketing by UAID if the user is not signed in but this is problematic since it means that a user can switch buckets if they sign in or out. (We may change the behavior of this bucketing scheme to simply disable the feature for users who are not signed in.)

Finally the bucketing value `'random'`, causes each request to be bucketed independently meaning that the same user will be in different buckets on different requests. This is typically used for features that should have no user-visible effects but where we want to ramp up something like the switch from master to shards or a new version of jquery.

The `'public_url_override'` property allows all requests, not just admin and internal requests, to turn on a feature and choose a variant via the `features` query param. Its value will almost always be true if it is present since it defaults to false if omitted.

Finally, two last shorthands:

First, a config stanza with only the key `'enabled'` and a string value can be replaced with just the string. So:

```
1 $server_config['foo'] = array('enabled' => 'on');
2 $server_config['bar'] = array('enabled' => 'off');
3 $server_config['baz'] = array('enabled' => 'some_variant');
```

Can be written simply:

---

```
1 $server_config['foo'] = 'on';
2 $server_config['bar'] = 'off';
3 $server_config['baz'] = 'some_variant';
```

And second, if a feature config is missing entirely, it's equivalent to specifying it as `'off'`. This allows dark changes to include code that checks for a feature before it has been added to `production.php`.

**Note for ops:** removing a feature config altogether, setting it to the string `'off'`, or setting `'enabled'` to `'off'` all completely disable the feature, ensuring that code guarded by `Feature::isEnabled` for that feature will never run. The best way to turn off an existing feature in an emergency would be to set `'enabled'` to `'off'`. To facilitate that, we should try to keep the `'enabled'` value on one line, whenever possible. Thus:

```
1 $server_config['foo'] = array(
2     'enabled' => array('foo' => 10, 'bar' => 10),
3 );
```

rather than

```
1 $server_config['foo'] = array(
2     'enabled' => array(
3         'foo' => 10,
4         'bar' => 10
5     ),
6 );
```

so that the bleary-eyed, junior ops person at 3am can do this:

```
1 $server_config['foo'] = array(
2     'enabled' => 'off', // array('foo' => 10, 'bar' => 10),
3 );
```

rather than this, which breaks the config file:

```
1 $server_config['foo'] = array(
2     'enabled' => 'off', // array(
3         'foo' => 10,
4         'bar' => 10
5     ),
6 );
```

Note, however, that removing the `'enabled'` property does mostly turn off the feature it doesn't completely disable it as it could still be enabled via an `'admin'` property, etc.

---

## Precedence:

The precedence of the various mechanisms for enabling a feature are as follows.

- If `'enabled'` is a string (variant name or `'off'`) the feature is entirely on or off for all requests.
- Otherwise, if the request is from an admin user or is an internal request, or if `'public_url_override'` is true and the request contains a `features` query param that specifies a variant for the feature in question, that variant is used. The value of the `features` param is a comma-delimited list of features where each feature is either simply the name of the feature, indicating the feature should be enabled with variant `'on'` or the name of a feature, a colon, and the variant name. E.g. a request with `features=foo,bar:x,baz:off` would turn on feature `foo`, turn on feature `bar` with variant `x`, and turn off feature `baz`.
- Otherwise, if the request is from a user specified in the `'users'` property, the specified variant is enabled.
- Otherwise, if the request is from a member of a group specified in the `'groups'` property the specified variant is enabled. (The behavior when the user is a member of multiple groups that have been assigned different variants is undefined. Beware nasal demons.)
- Otherwise, if the request is from an admin, the `'admin'` variant is enabled.
- Otherwise, if the request is an internal request, the `'internal'` variant is enabled.
- Otherwise, the request is bucketed and a variant is chosen so that the correct percentage of bucketed requests will see each variant.

## Errors

There are a few ways to misuse the Feature API or misconfigure a feature that may be detected and logged. (Some of these are not currently detected but may be in the future.)

1. Calling `Feature::variant` for a single-variant feature.
2. Calling `Feature::variant` in code not guarded by an `Feature::isEnabled` check.
3. Including `'on'` as a variant name in a multi-variant feature.
4. Setting `'enabled'` to numeric value less than 0 or greater than 100.
5. Setting the percentage value of a variant in `'enabled'` to a value less than 0 or greater than 100.
6. Setting `'enabled'` such that the sum of the variant percentages is greater than 100.



- 
7. Setting `'enabled'` to a non-numeric, non-string, non-array value.
  8. When `'enabled'` is an array, setting the `'users'` or `'groups'` property to an array that includes a key that is not a key in `'enabled'`.
  9. When `'enabled'` is an array, setting the `'admin'` or `'internal'` property to a value that is not a key in `'enabled'`.

## The life cycle of a feature

The Feature API was designed with a eye toward making it a bit easier for us to push features through a predictable life cycle wherein a feature can be created easily, ramped up, A/B tested, and then cleaned up, either by being promoted to a full-fledged feature flag, by removing the configuration and associated feature checks but keeping the code, or deleting the code altogether.

The basic life cycle of a feature might look like this:

1. Developer writes some code guarded by `Feature::isEnabled` checks. In order to test the feature in development they will add configuration for the feature to `development.php` that turns it on for specific users or admin or sets `'enabled'` to 0 so they can test it with a URL query param.
2. At some point the developer will add a config stanza to `production.php`. Initially this may just be a place holder that leaves the feature entirely disabled or it may turn it on for admin, etc.
3. Once the feature is done, the `production.php` config will be changed to enable the feature for a small percentage of users for an operational smoke test. For a single-variant feature this means setting `'enabled'` to a small numeric value; for a multi-variant feature it means setting `'enabled'` to an array that specifies a small percentage for each variant.
4. During the rampup period the percentage of users exposed to the feature may be moved up and down until the developers and ops folks are convinced the code is fully baked. If serious problems arise at any point, the new code can be completely disabled by setting enabled to `'off'`.
5. If the feature is going to be part of an A/B experiment, then the developers will (working with the data team) figure out the best percentage of users to expose the feature to and how long the experiment will have to run in order to gather good experimental data. To launch the experiment the production config will be changed to enable the feature or its variants for the appropriate percentage of users. After this point the percentages should be left alone until the experiment is complete.

---

At this point there are a number of things that can happen: if the experiment revealed a clear winner we may simply want to keep the code, possibly putting it under control of a top-level feature flag that ops can use to disable the feature for operational reasons. Or we may want to discard all the code related to the feature. Or we may want to run another experiment based on what we learned from this one. Here's what will happen in those cases:

### **To keep the feature as a permanent part of the web site without creating a top-level feature flag**

1. Change the value of the feature config to the name of the winning variant ('on' for a single-variant feature).
2. Delete any code that implements other variants and remove the calls to `Feature::variant` and any related conditional logic (e.g. switches on the variant name).
3. Remove the `Feature::isEnabled` checks but keep the code they guarded.
4. Remove the feature config.

### **To keep a feature under the control of a full-fledged feature flag. (I.e. for things that will typically be enabled but which we want to preserve the ability to turn off with a simple config change.)**

1. Change the value of the feature config to the name of the winning variant ('on' for a single-variant feature).
2. Delete any code that implements other variants and remove the calls to `Feature::variant` and any related conditional logic (e.g. switches on the variant name).
3. Add a new config named with a `feature_` prefix and set its value to 'on'.
4. Change all the `Feature::isEnabled` checks for the old flag name to the new feature flag.
5. Remove the old config.

### **To remove a feature all together**

1. Change the value of the feature config to 'off'.
2. Delete all code guarded by `Feature::isEnabled` checks and then remove the checks.
3. Remove the feature config.

---

## To run a new experiment based on the same code

1. Set the enabled value of the feature config to `'off'`.
2. Create a new feature config with a similar name but suffixed with `_vN` where N is 2 if this is the second experiment, 3 if is the third. Set it to `'off'`.
3. Change all the `Feature::isEnabled` checks for the old feature to the new feature.
4. Delete the old config.
5. Implement the changes required for the new experiment, deleting old variants and adding new ones as needed.
6. Rampup and then A/B test the new feature as normal.
7. Promote, cleanup, or re-experiment as appropriate.

## A few style guidelines

To make it easier to push features through this life cycle there are a few coding guidelines to observe.

First, the feature name argument to the Feature methods (`isEnabled`, `variant`, `isEnabledFor`, and `variantFor`) should always be a string literal. This will make it easier to find all the places that a particular feature is checked. If you find yourself creating feature names at run time and then checking them, you're probably abusing the Feature system. Chances are in such a case you don't really want to be using the Feature API but rather simply driving your code with some plain old config data.

Second, the results of the Feature methods should not be cached, such as by calling `Feature::isEnabled` once and storing the result in an instance variable of some controller. The Feature machinery already caches the results of the computation it does so it should already be plenty fast to simply call `Feature::isEnabled` or `Feature::variant` whenever needed. This will again aid in finding the places that depend on a particular feature.

Third, as a check that you're using the Feature API properly, whenever you have an if block whose test is a call to `Feature::isEnabled`, make sure that it would make sense to either remove the check and keep the code or to delete the check and the code together. There shouldn't be bits of code within a block guarded by an `isEnabled` check that needs to be salvaged if the feature is removed.