Everyone can fly a flag

Enabling collaboration over the full lifecycle of a feature

To start off, how many of us are familiar with feature flags?

Put simply, feature flags are simply on/off switches. In our Django/Wagtail world, they can enable or disable functionality in code or blocks of HTML in templates.

I'm going to start with some in-code examples at first, because my intent is to contrast that need to manage feature flags in code with allowing non-developers to manage them.

```
# settings.py
MY_FLAG = True
```

Implementing feature flags can be as simple as setting a variable in the Django settings file to true or false

```python
# views.py
def my_view(request):
    if settings.MY_FLAG:
        return render(request, "new.html")
    return render(request, "old.html")
```

And then using that variable in, say, a Django view to determine which template to use.

But what if you want something more complex?

```
# settings/testing.py
MY_FLAG = True

# settings/prod.py
MY_FLAG = False
```

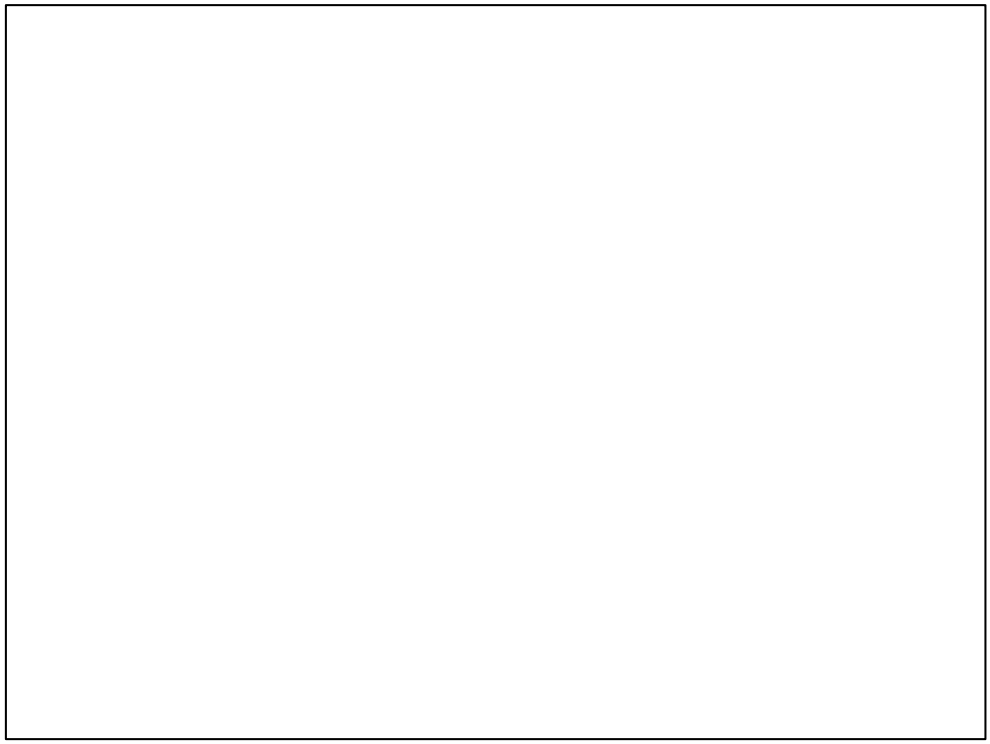
Maybe you want to be able to turn on a flag in your testing environment but not in production.

```python
# settings.py
MY_FLAG = (
    os.environ.get('ENV') == 'test'
)
```

Maybe you want to be able to turn on a flag in your testing environment but not in production.

```python
# settings.py
if date.today() > date(2019, 07, 28):
    MY_FLAG = True
else:
    MY_FLAG = False
```

But maybe you want to the flag to automatically enable after a specific date and time



Now what if you have a lot of flags. This can start getting complicated.

And now, let me throw in one more question. What if you want to able to enable or disable flags without a deployment? Then you'll need database models and a way to manage the flags in the Django admin.

```python
# models.py
from django.db import models

class FeatureFlag(models.Model):
    name = models.CharField(max_length=64)
    on = models.BooleanField()
```

Oh, but we needed to be able to set a date after which it'll be turned on:

```python
# models.py
from django.db import models

class FeatureFlag(models.Model):
    name = models.CharField(max_length=64)
    on = models.BooleanField()
    date = models.DateField()
```

And we had that environment check too.

```python
# models.py
from django.db import models

class FeatureFlag(models.Model):
    name = models.CharField(max_length=64)
    on = models.BooleanField()
    date = models.DateField()
    env = models.CharField(max_length=64)
```

And this is before we've created the model form or admin or logic that will determine what the state might actual be.

This is about the time most of us would realize we should look for a library.

This is basically how feature flags at CFPB (and I imagine, many other places).

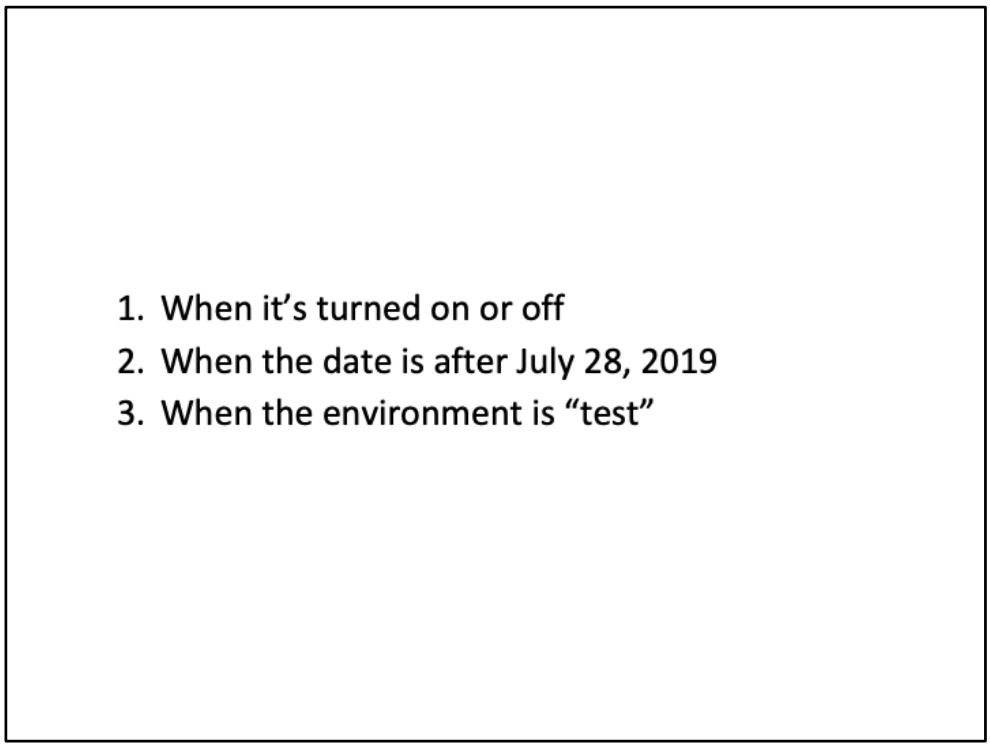
What we started using as simple ways to toggle features and work in progress have become a tool a rely on in more and more complex situations.

For that reason, and because at the time we couldn't find anything that was really comparable to what we were trying to do, we broke off our feature flag app into a separate library, Django-Flags.

So, what does it do?

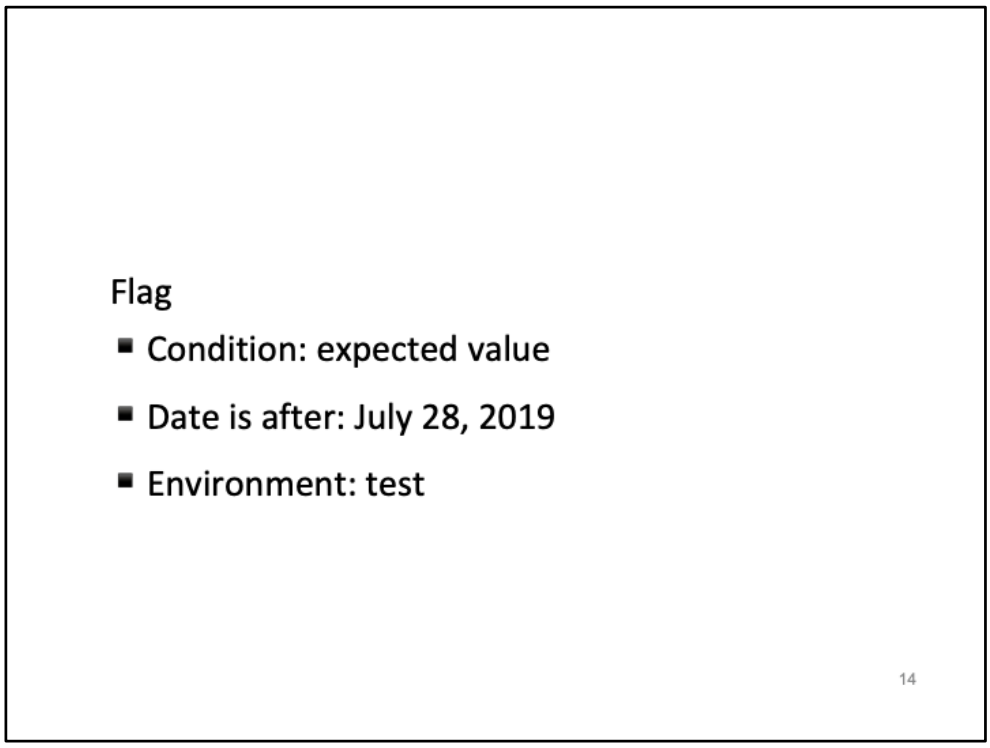
Django-Flags supports defining feature flags in settings and in the database, like the examples I've given so far.

It has utility functions and decorators for checking if a flag is enabled or disabled in Django and Jinja2 templates and in code.



It also the concept of "conditions". Earlier we had three ways a feature flag could be enabled.

Each of these had an expected value, and the last two had values we could test against to see if they matched that expected value.



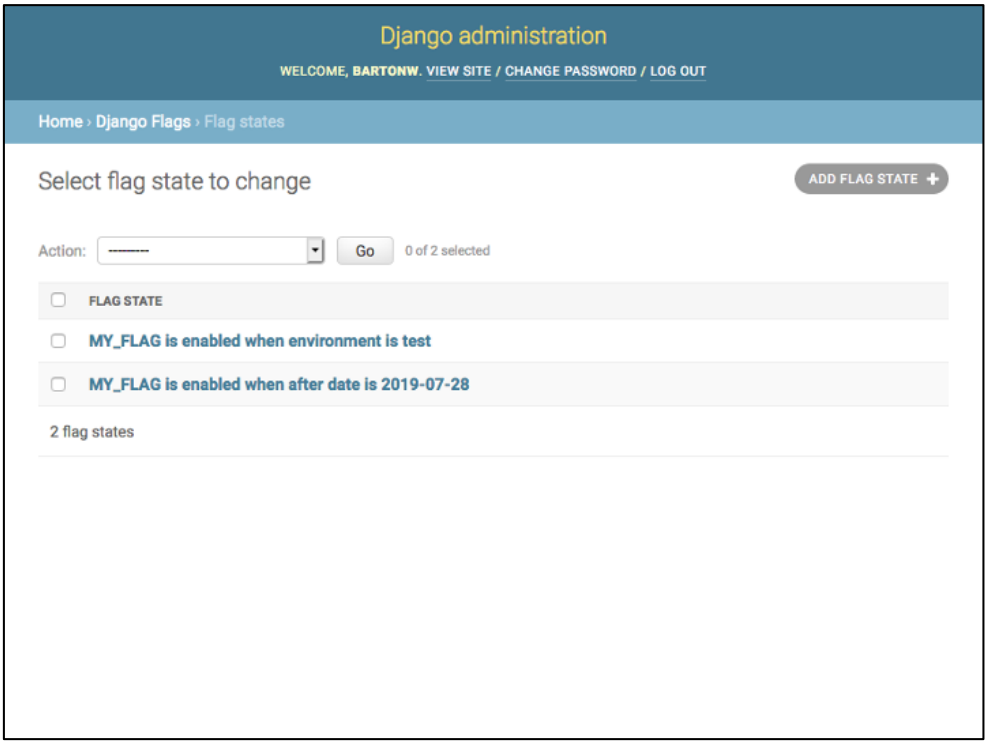
This is how Django-Flags conceptualizes feature flags: Flags are a name with a set of conditions.

Each of those conditions has an expected value.

When the flag's state is checked each condition's expected value is checked against the current value. And that's how we determine whether the flag should be enabled or not.

```
# settings.py
FLAGS = {
    'MY_FLAG': [
        {'condition': 'date',
         'value': '2019-07-28'},
        {'condition': 'environment',
         'value': 'test'},
    ]
}
```

We can replicate the date and environment conditions from the previous examples using these settings:



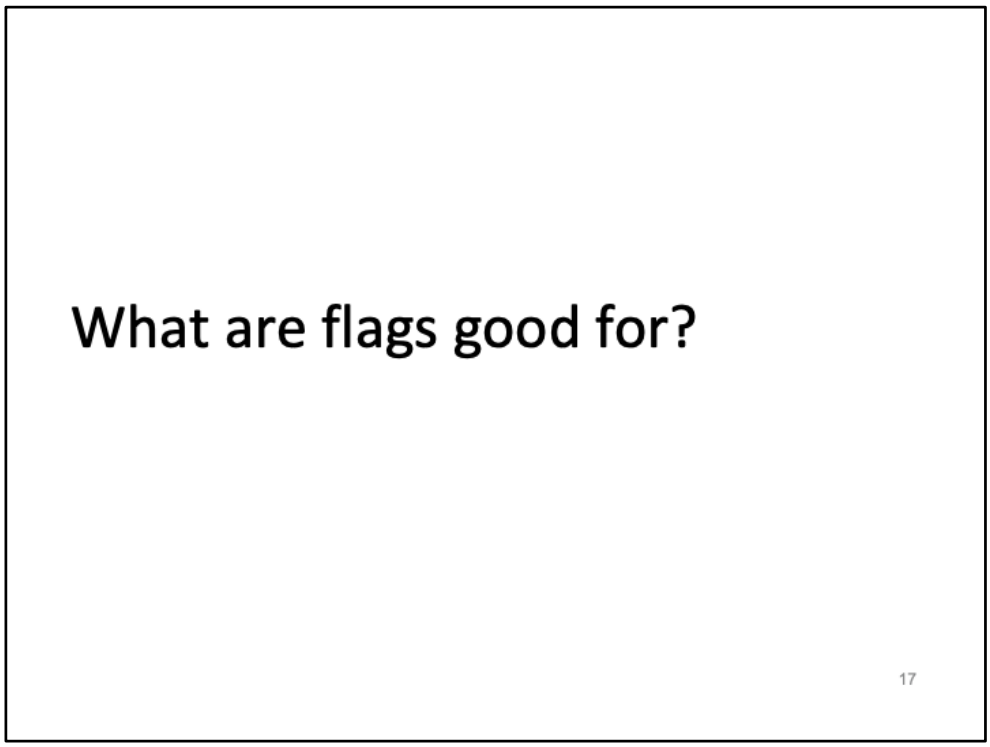
Or in the admin:

Because conditions are extensible (you simply register a function that evaluates a request or other set of arguments), we have a lot of flexibility in terms of what can enable a flag.

One thing to note, is that flag can have multiple conditions and if any of those conditions are met the flag is enabled.

So will enable the flag for anyone after July 28, 2019. But it is always enabled in the “test” environment.

Everyone with me so far?



At CFPB we've used feature flags to enable collaboration between developers, designers, stakeholders and subject matter experts over the whole life-cycle of a feature or project.

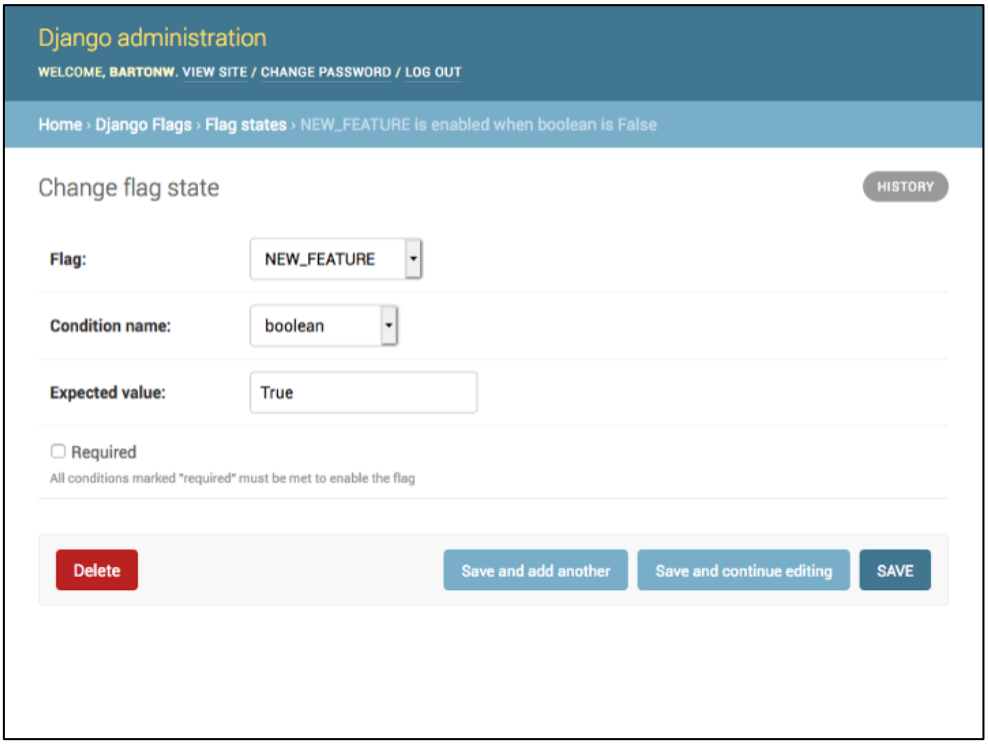
At the start of the project we'll add a feature flag for it. For example,

```
# settings.py
FLAGS = {
    'NEW_FEATURE': []
}
```

At the start of the project we'll add a feature flag for it.

```python
# urls.py
from flags.urls import flagged_path
urlpatterns = [
    flagged_path(
        'NEW_FEATURE',
        'feature/',
        serve_regulations
    ),
]
```

And use that flag in code.



And then enable that flag on a development server.

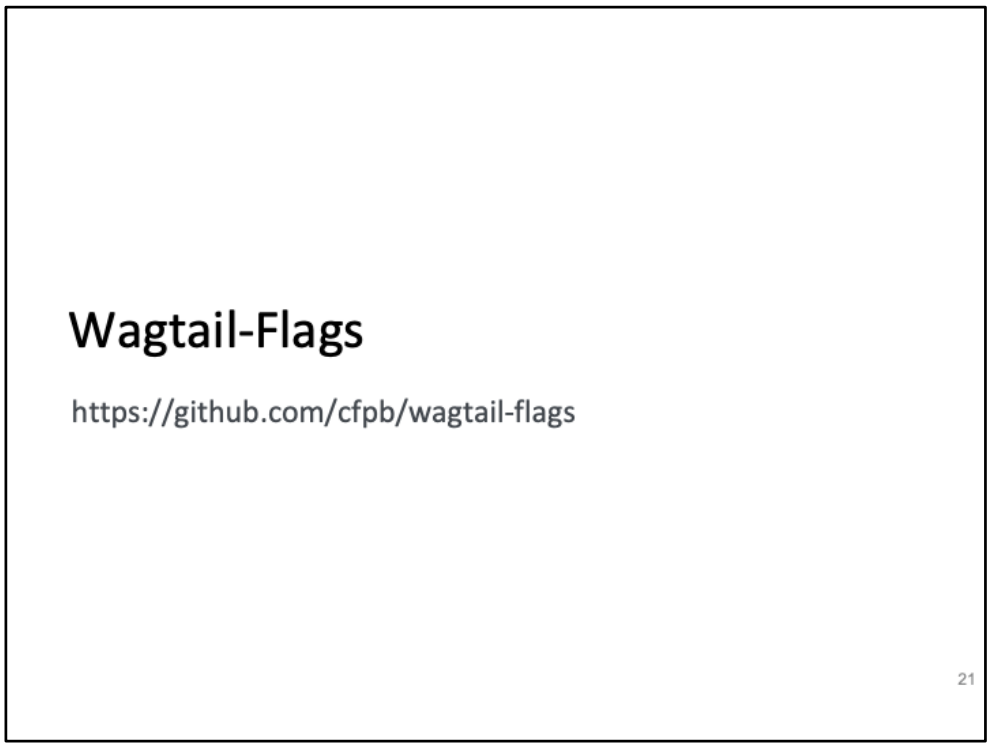
Once we've done that, our subject matter experts and stakeholders can review the work in real time, closing that feedback loop and allowing us be more productive (and therefore good stewards of the public trust).

More importantly though, because this gets us closer to real-time collaboration between different domains of subject matter expertise (between web development and market regulation, for example), it allows us to build a good working relationship with our colleagues.

One thing you'll notice is that enabling the flag does not require a developer. It can be performed by anyone with appropriate access to the Django admin.

But, because of how the Django model admin mirrors the model schema, the Django-Flags admin is still fairly specialized. It's still quite "developery" for lack of a better word.

So how do we (speaking for myself as a developer) better enable our non-developer colleagues to manage the flags?



This is basically how feature flags at CFPB (and I imagine, many other places).

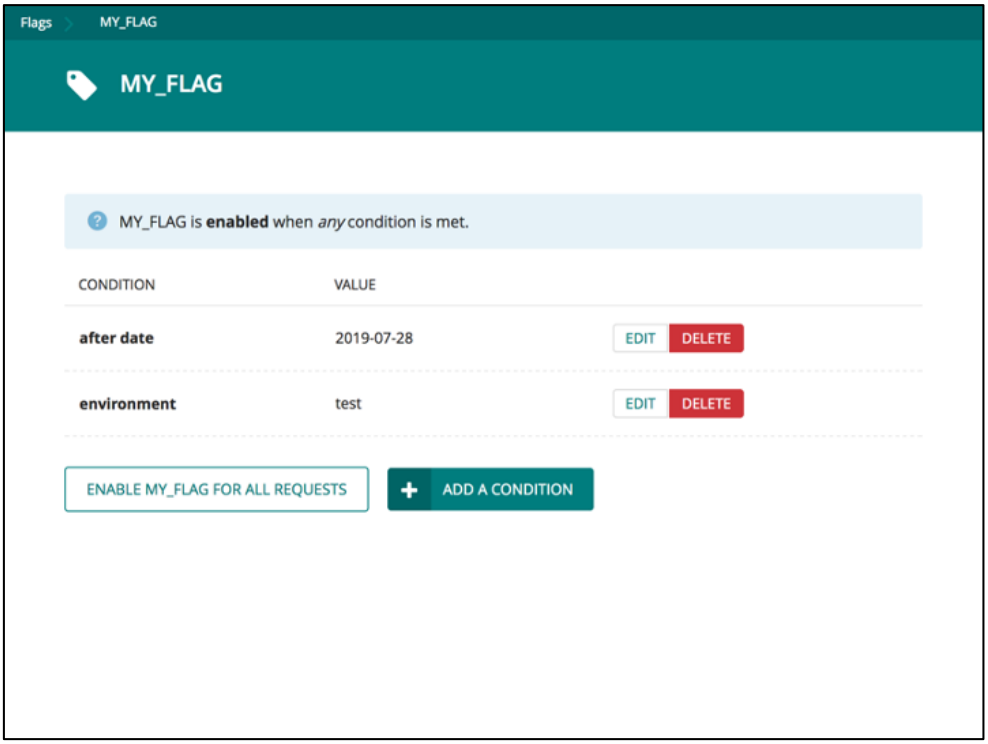
What we started using as simple ways to toggle features and work in progress have become a tool a rely on in more and more complex situations.

For that reason, and because at the time we couldn't find anything that was really comparable to what we were trying to do, we broke off our feature flag app into a separate library, Django-Flags.

So, what does it do?

Django-Flags supports defining feature flags in settings and in the database, like the examples I've given so far.

It has utility functions and decorators for checking if a flag is enabled or disabled in Django and Jinja2 templates and in code.

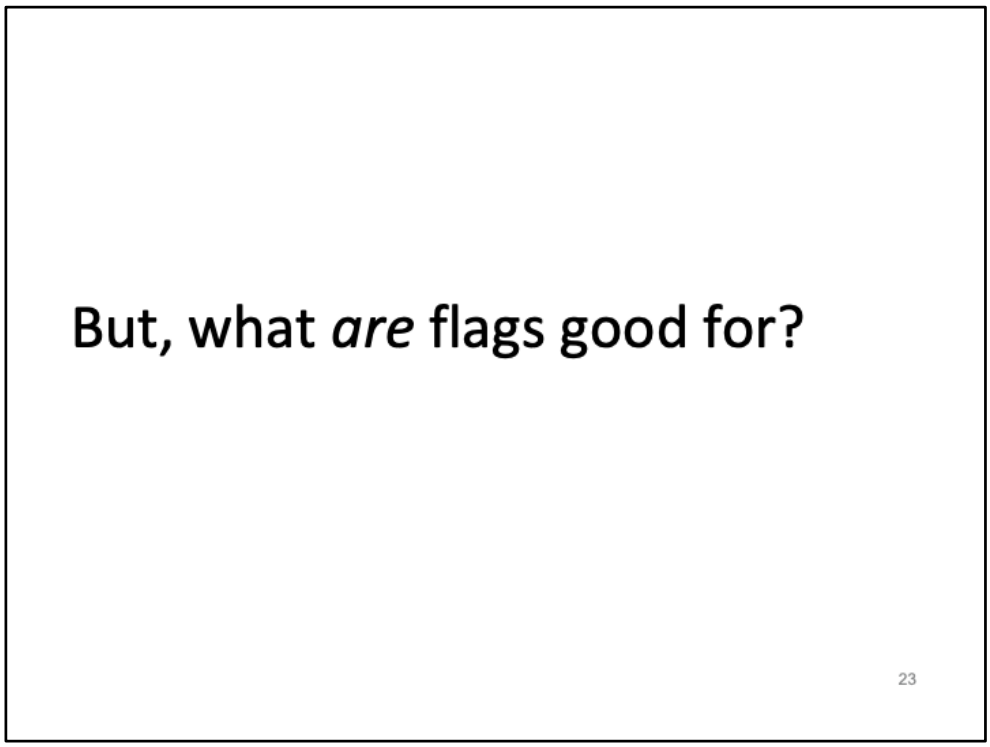


This means that our content editors (who tend to be the most in-touch with our stakeholders) can enable or disable a flag if they need to.

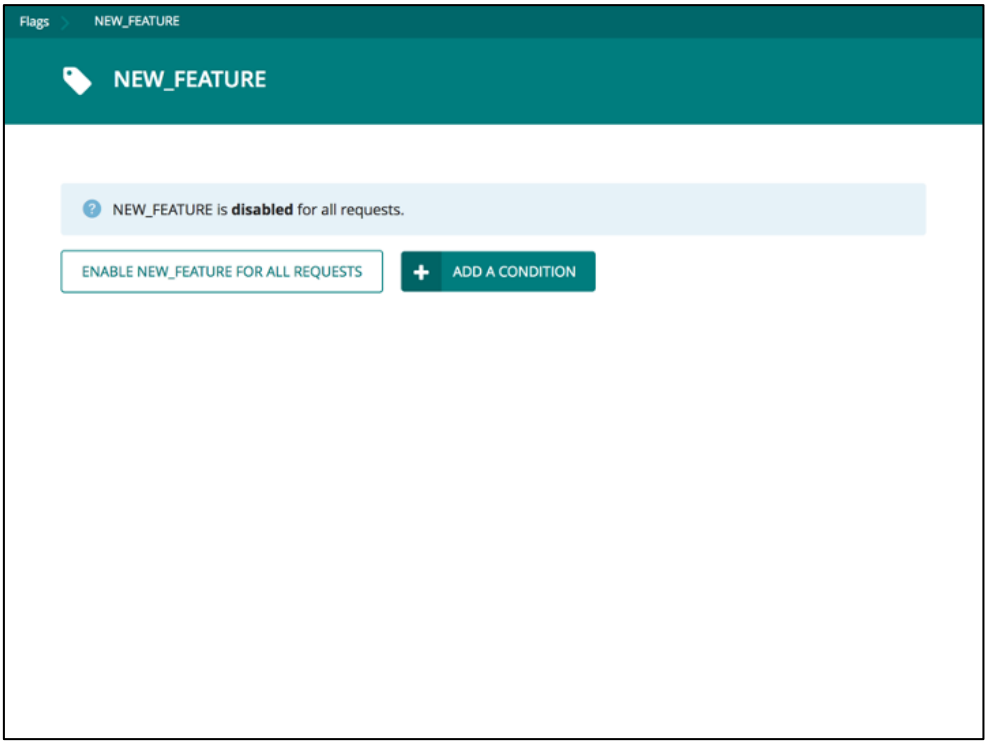
We have human-readable (and, hopefully, comprehensible) descriptions of the flag's state, and easy buttons for enabling and disabling flags.

The enable/disable buttons create and edit boolean conditions that are true or false.

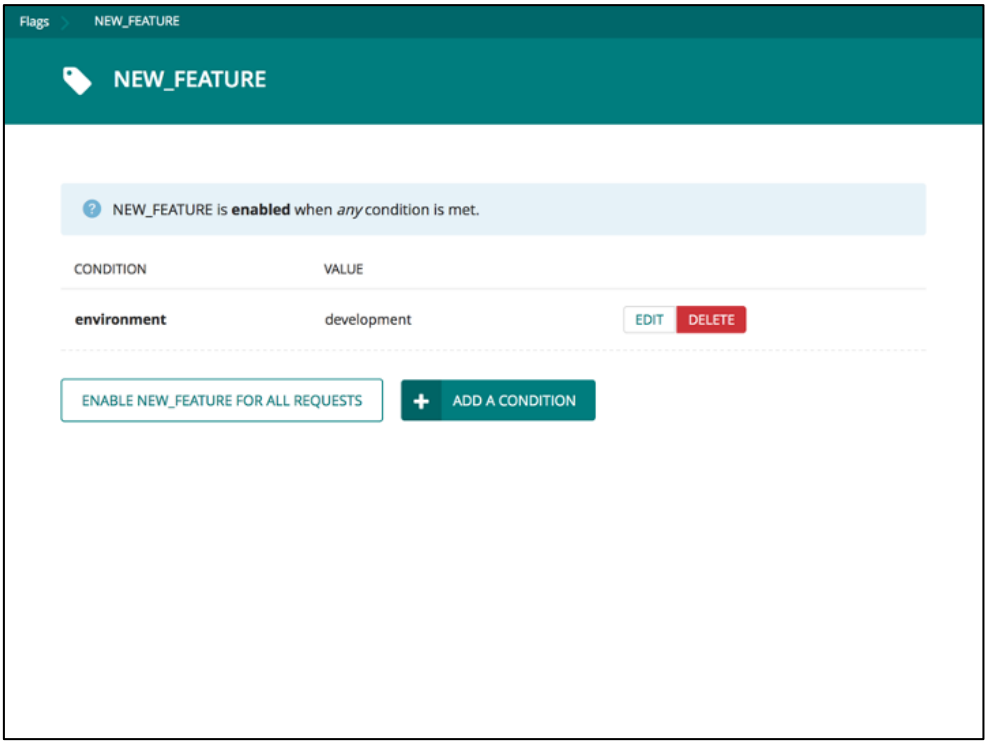
And you can add, edit, and delete conditions, as you might expect to be able to.



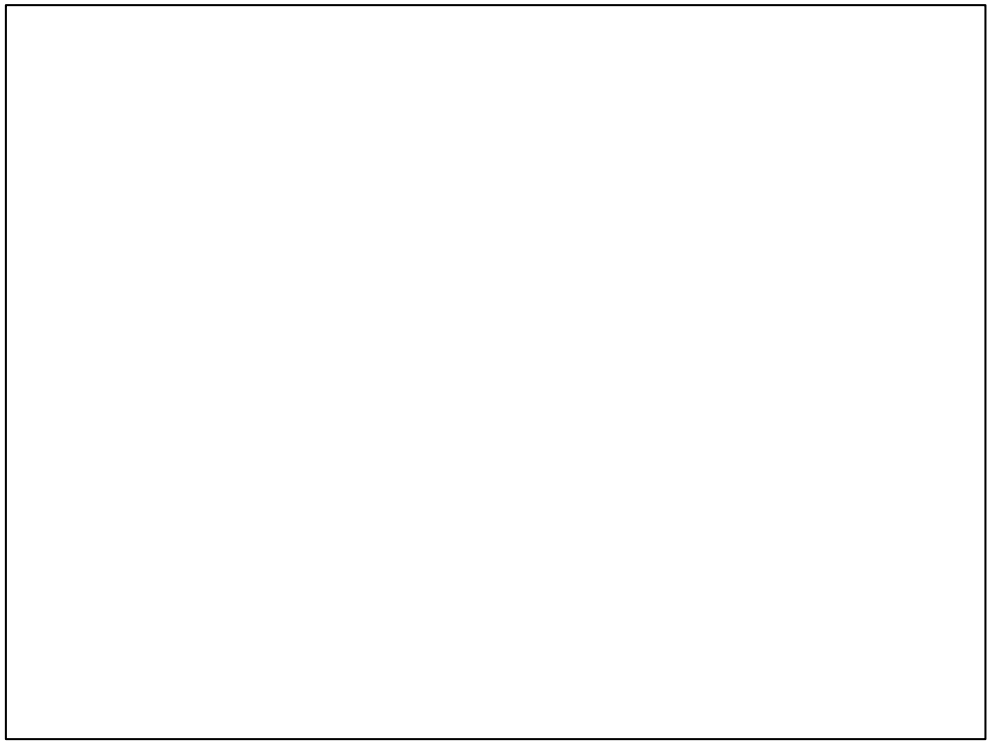
Getting back to our flag lifecycle question, what are they good for?



We have the flag we created when we started working on the feature.

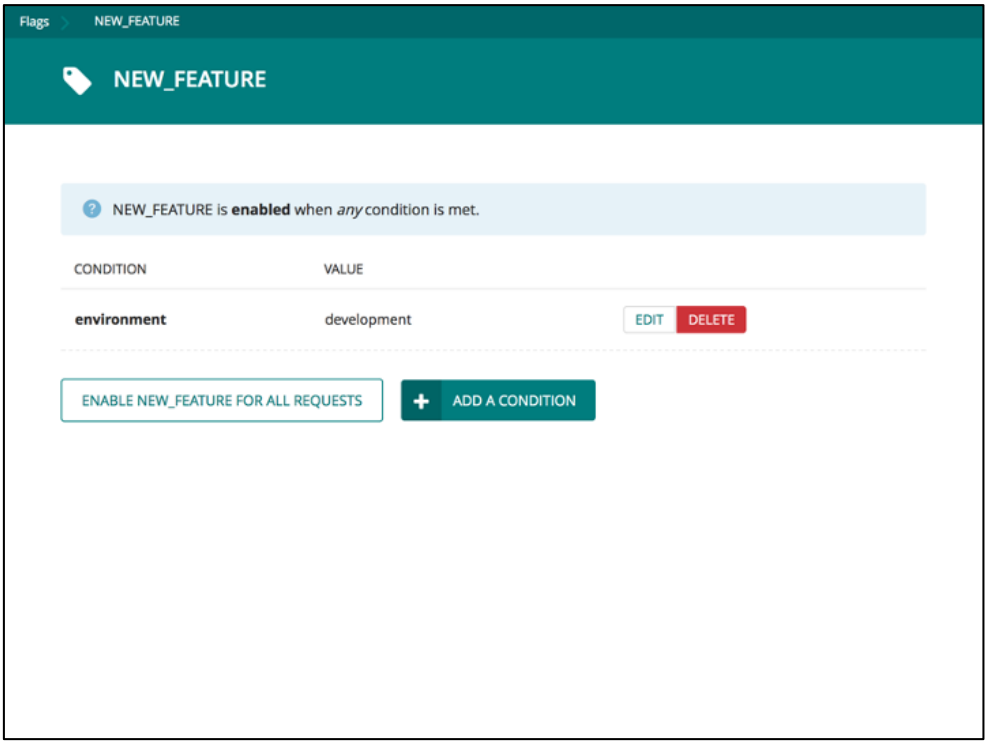


We can enable it on a development server so that everyone can see the feature as it exists in code right now.

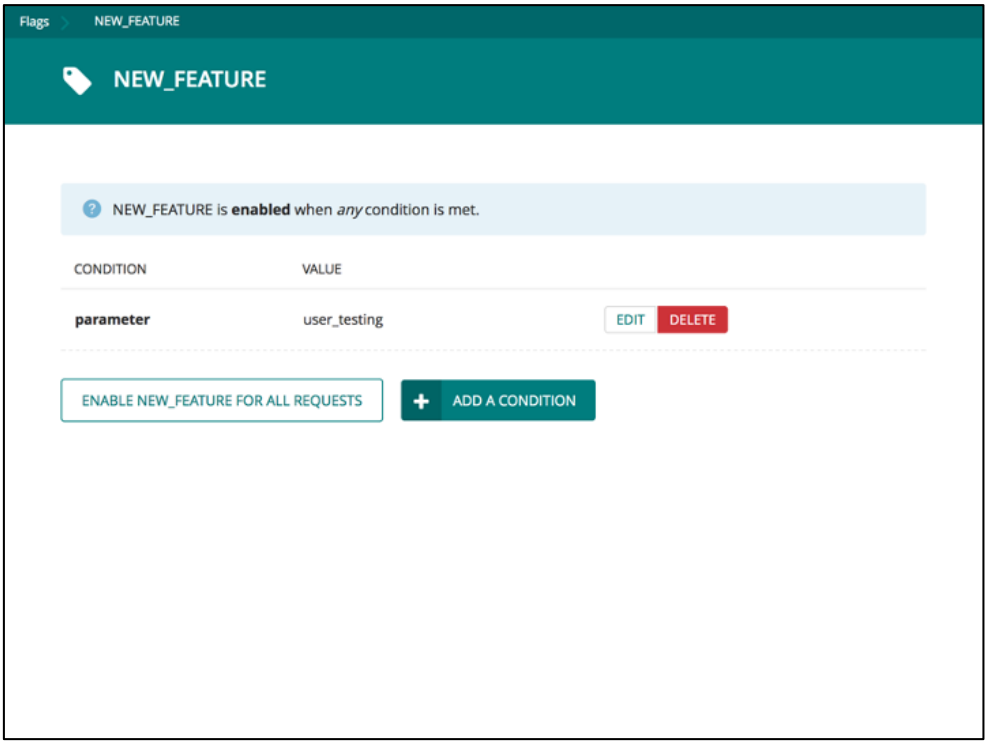


That same flag can be used when we're ready to do user testing.

Our UX researchers can enable the flag for a particular set of conditions in which user testing will occur.



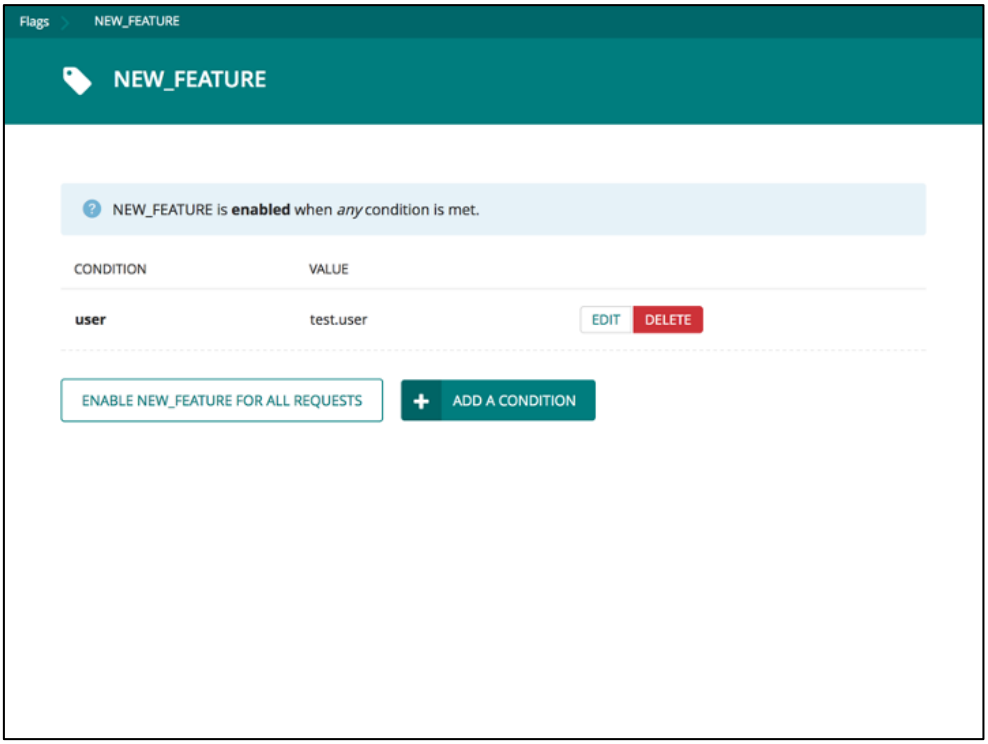
Maybe we have a specific environment to test in, so the feature will always be enabled there.



Or we can add require a GET parameter to the URLs to enable the feature and test it with it.

```
https://testing.server/?user_testing
```

And then visit our test server and provide that parameter.

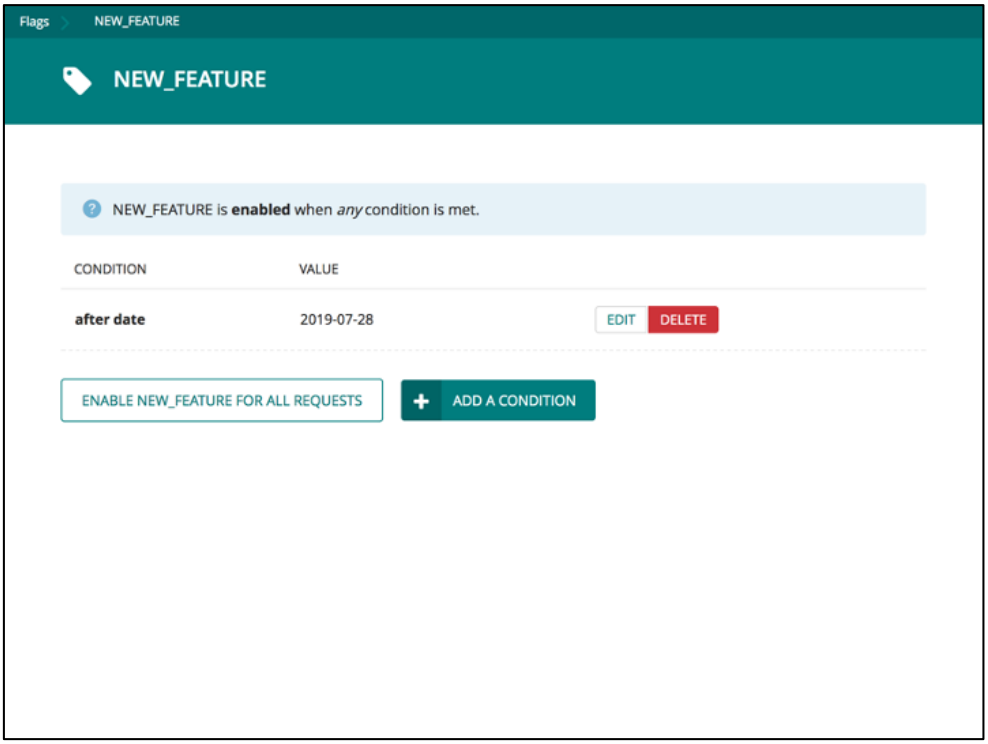


Or maybe we have a particular user that could log in for testing, so that the feature will always be enabled for that user.

```python
# urls.py
urlpatterns = [
    flagged_path(
        'NEW_FEATURE',
        'feature/',
        serve_new_feature,
        fallback=serve_old_feature,
    ),
]
```

Our new feature might be replacing an existing feature.

So we can sunset that feature at the same time we launch the new one.



Then, when we're ready to launch the feature, it might be part of a larger initiative, so it we might be coordinating it with events, blog posts, etc, that have scheduled publishing dates.

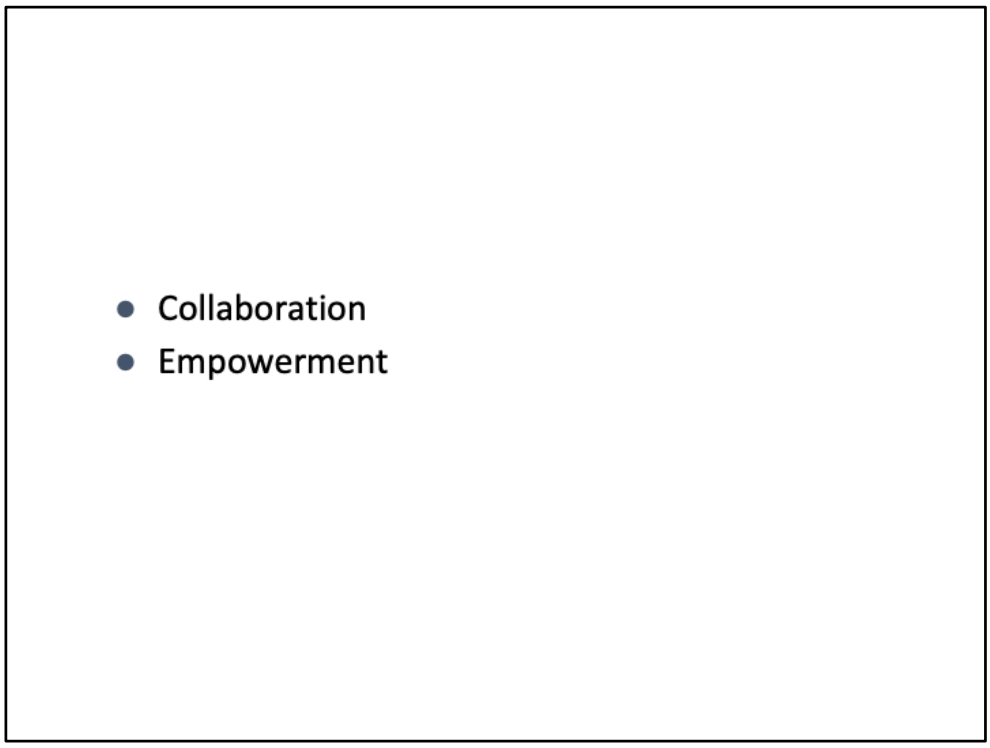
The content editors can simply set a date condition on the flag in production the same as they set on the blog post announcing the feature and let it enable itself when the time comes.

There’s no morning-of-launch crunch time to get it deployed because it’s already in production, just waiting to turn itself on.

And if the launch is delayed (maybe printed material that goes with it isn't done or there's some policy delay), content editors have the freedom to make the change without waiting on a developer.

In this way we've got developers, designers, stakeholders, UX researchers, and content editors are all involved and empowered throughout the lifecycle of a feature. No one is waiting for anyone else to enable something and everyone is trusted and able to do all the things they may need to within their domain of expertise.

As they should be.



And that's the real power of feature flags — collaboration and empowerment.

Thank you.

Thank you!