Stability without Stagnation

Learning from Ember.js

"Stability without Stagnation" postulates that there's something we can do to deal with the tension. It stands in stark contrast to another meme that has a lot of power in the tech community: Move Fast and Break Things, which is the idea that we have no choice but to accept instability as the price of progress.

But ultimately, even the creator of Move Fast and Break Things realized that you can do both at the same time.

"It may not be quite as catchy as 'move fast and break things.' But it's how we operate now."

–Mark Zuckerberg

Move Fast and Break Things is a catchy slogan, and the idea of "we have to accept instability in order to make progress" is often repeated.

“What we realized over time is that it wasn't helping us to move faster because we had to slow down to fix these bugs and it wasn't improving our speed"

–Mark Zuckerberg

But instability is a drag on innovation.

SKYLIGHT

I work on a product called Skylight. It's a performance monitoring tool for Rails applications.

The Log-Normal Reality

blog.skylight.io/the-log-normal-reality/

Get Skylight

29 Oct 2015

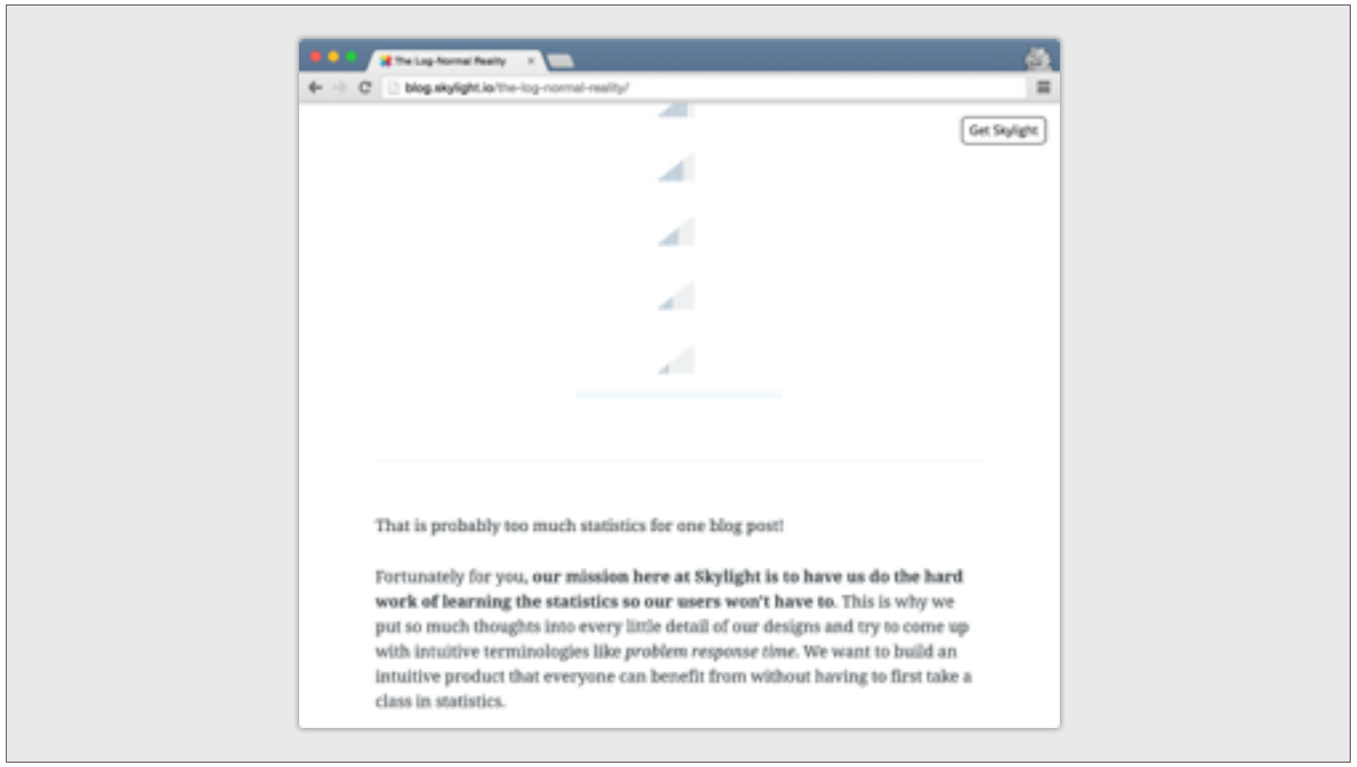
The Log-Normal Reality

I took some time off in October to visit my friends and family in Hong Kong. When I returned from my vacation, I was very excited to see that we had shipped our new design for the endpoints page.

If you haven't seen it yet, go ahead and see it in action with data from your own apps! Here is what it looks like in our app:

Direwolf!

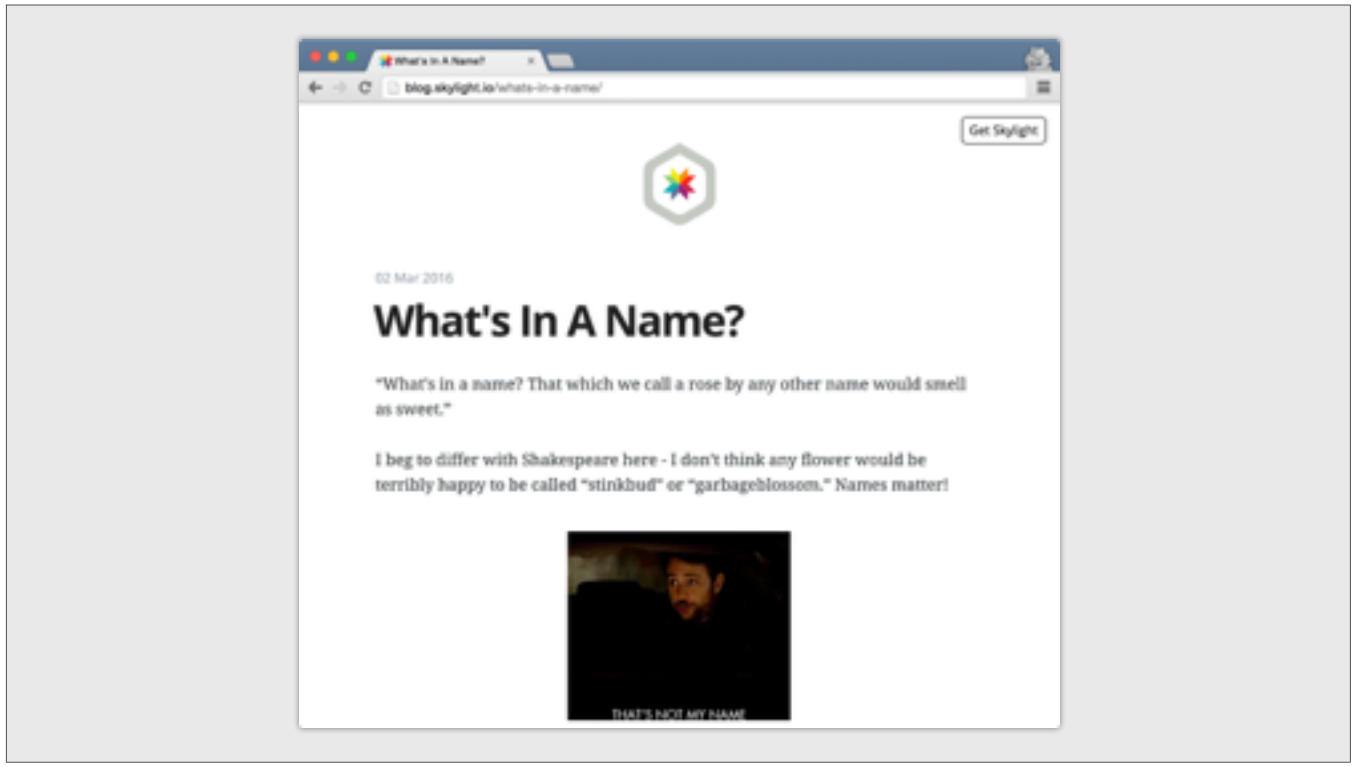
One thing I really love about working on Skylight is that we spend a lot of time on getting even seemingly trivial things right.



That is probably too much statistics for one blog post!

Fortunately for you, **our mission here at Skylight is to have us do the hard work of learning the statistics so our users won't have to.** This is why we put so much thoughts into every little detail of our designs and try to come up with intuitive terminologies like *problem response time*. We want to build an intuitive product that everyone can benefit from without having to first take a class in statistics.

This blog post, (which had a lot of math in it), ending up being about what icon we should use to represent "popularity" of an endpoint. We spent weeks on this project, and I love this work.



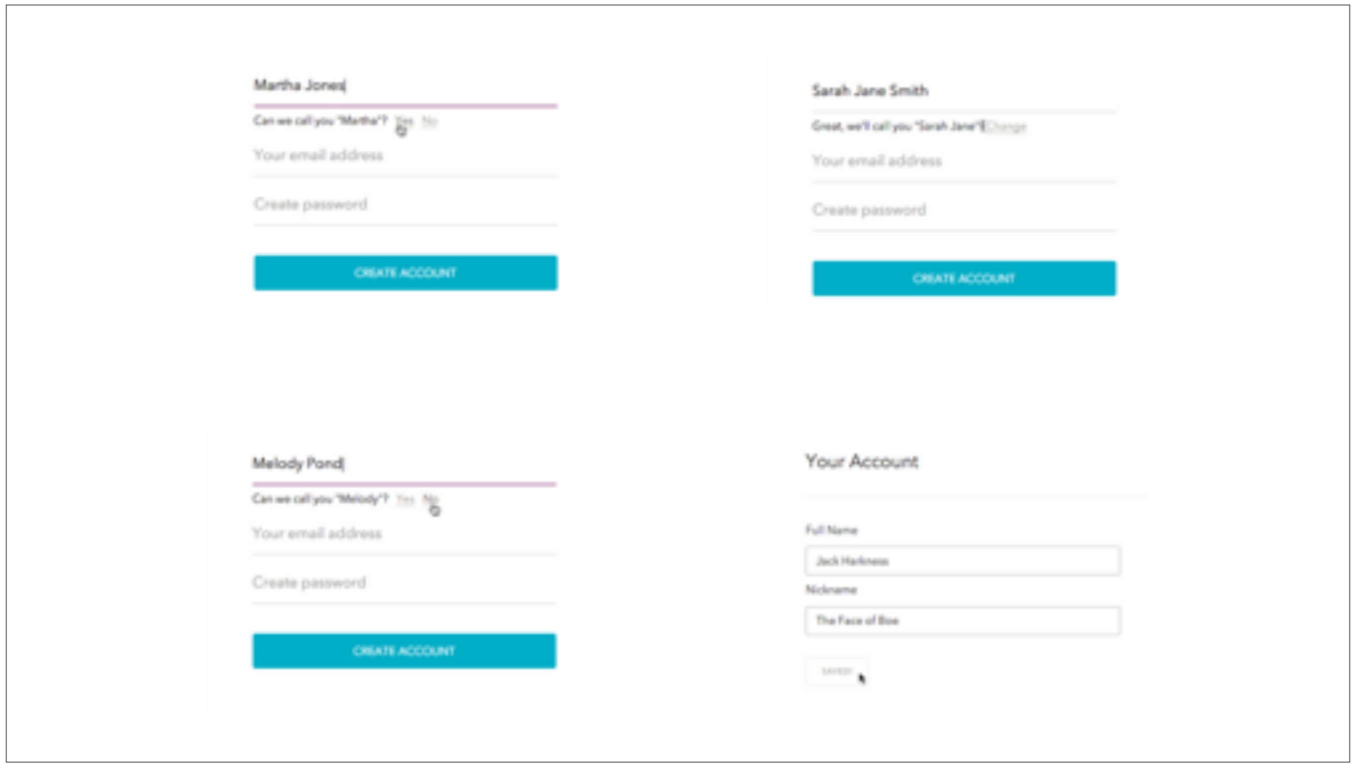
02 Mar 2016

What's In A Name?

"What's in a name? That which we call a rose by any other name would smell as sweet."

I beg to differ with Shakespeare here - I don't think any flower would be terribly happy to be called "stinkbud" or "garbageblossom." Names matter!

Another example: we needed to support changing the user's name. It turns out that names are way more involved than you think.


Martha Jones

Can we call you "Martha"? Yes No

Sarah Jane Smith

Great, we'll call you "Sarah Jane"! Change

Melody Pond

Can we call you "Melody"? Yes No

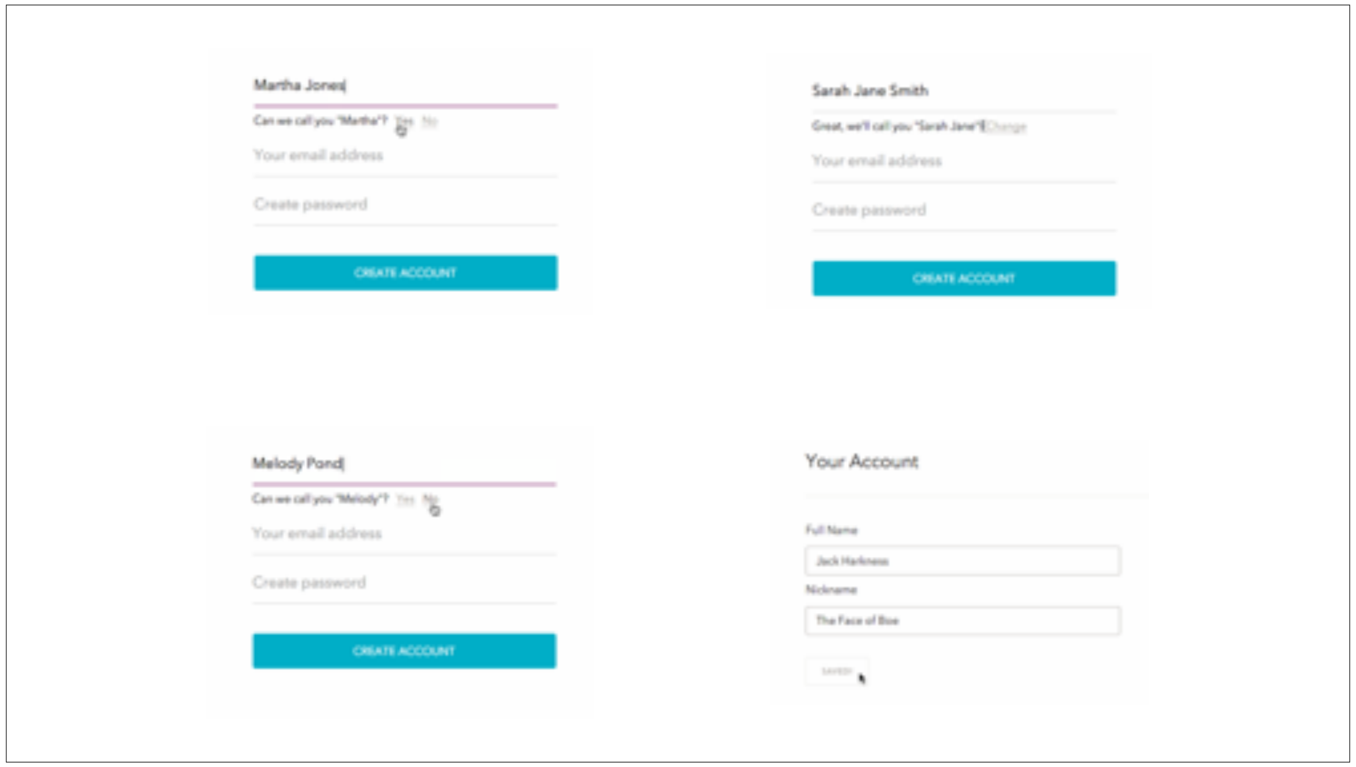
Your Account

Full Name: Jack Harkness

Nickname: The Face of Boe

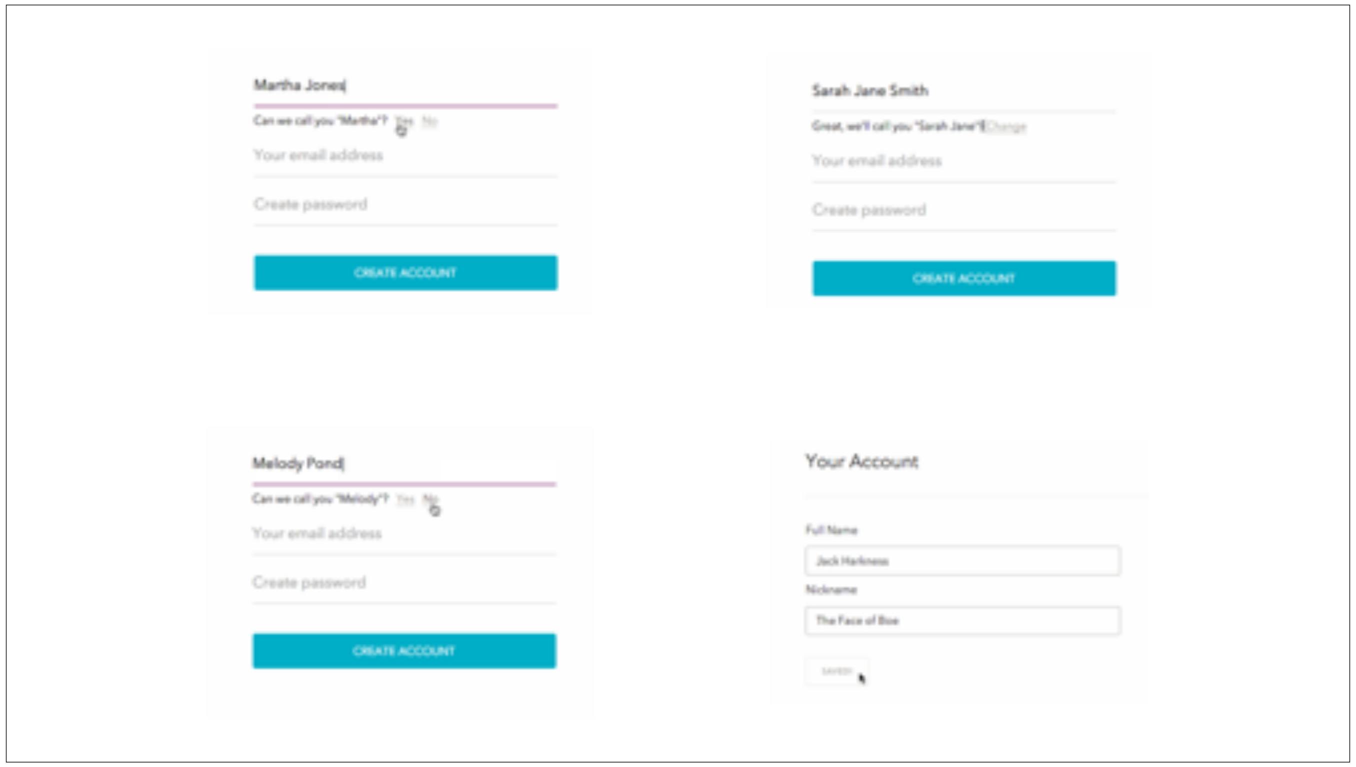
We ended up with a really nice system where we try to infer the user's nickname, but give them a low-friction way to confirm or deny the nickname. One subtle thing that we did was remember whether the user actually confirmed the nickname or whether it was just inferred. Once a customer has chosen their own nickname or agreed to be called by their first name, we switch this field to true and fine-tune our heuristics to make this choice extra sticky.

We collectively spent more than a week on implementing this feature, and I loved every minute of it.



We ended up with a really nice system where we try to infer the user's nickname, but give them a low-friction way to confirm or deny the nickname. One subtle thing that we did was remember whether the user actually confirmed the nickname or whether it was just inferred. Once a customer has chosen their own nickname or agreed to be called by their first name, we switch this field to true and fine-tune our heuristics to make this choice extra sticky.

We collectively spent more than a week on implementing this feature, and I loved every minute of it.


Martha Jones

Can we call you "Martha"? Yes No

Sarah Jane Smith

Great, we'll call you "Sarah Jane"! Change

Melody Pond

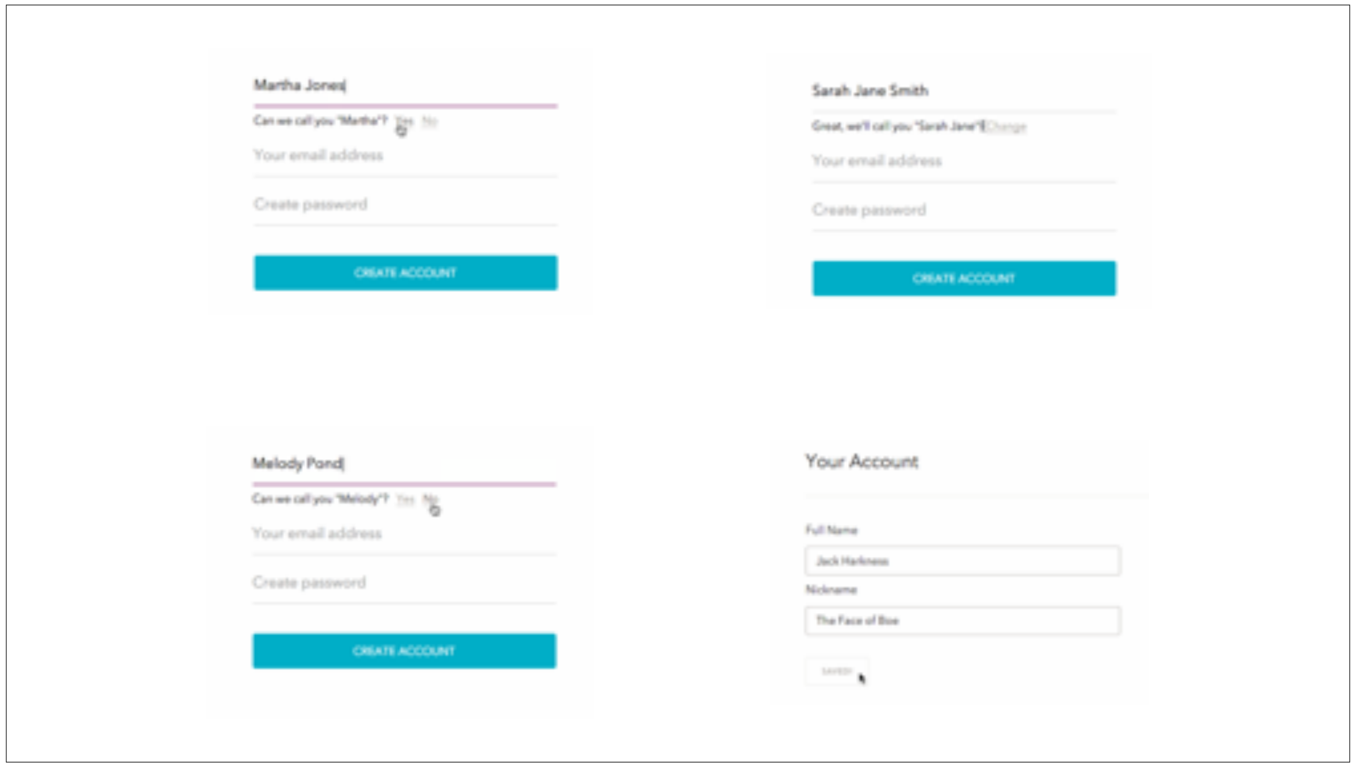
Can we call you "Melody"? Yes No

Full Name: Jack Harkness

Nickname: The Face of Boe

We ended up with a really nice system where we try to infer the user's nickname, but give them a low-friction way to confirm or deny the nickname. One subtle thing that we did was remember whether the user actually confirmed the nickname or whether it was just inferred. Once a customer has chosen their own nickname or agreed to be called by their first name, we switch this field to true and fine-tune our heuristics to make this choice extra sticky.

We collectively spent more than a week on implementing this feature, and I loved every minute of it.


Martha Jones

Can we call you "Martha"? Yes No

Your email address

Create password

Sarah Jane Smith

Great, we'll call you "Sarah Jane"! Change

Your email address

Create password

Melody Pond

Can we call you "Melody"? Yes No

Your email address

Create password

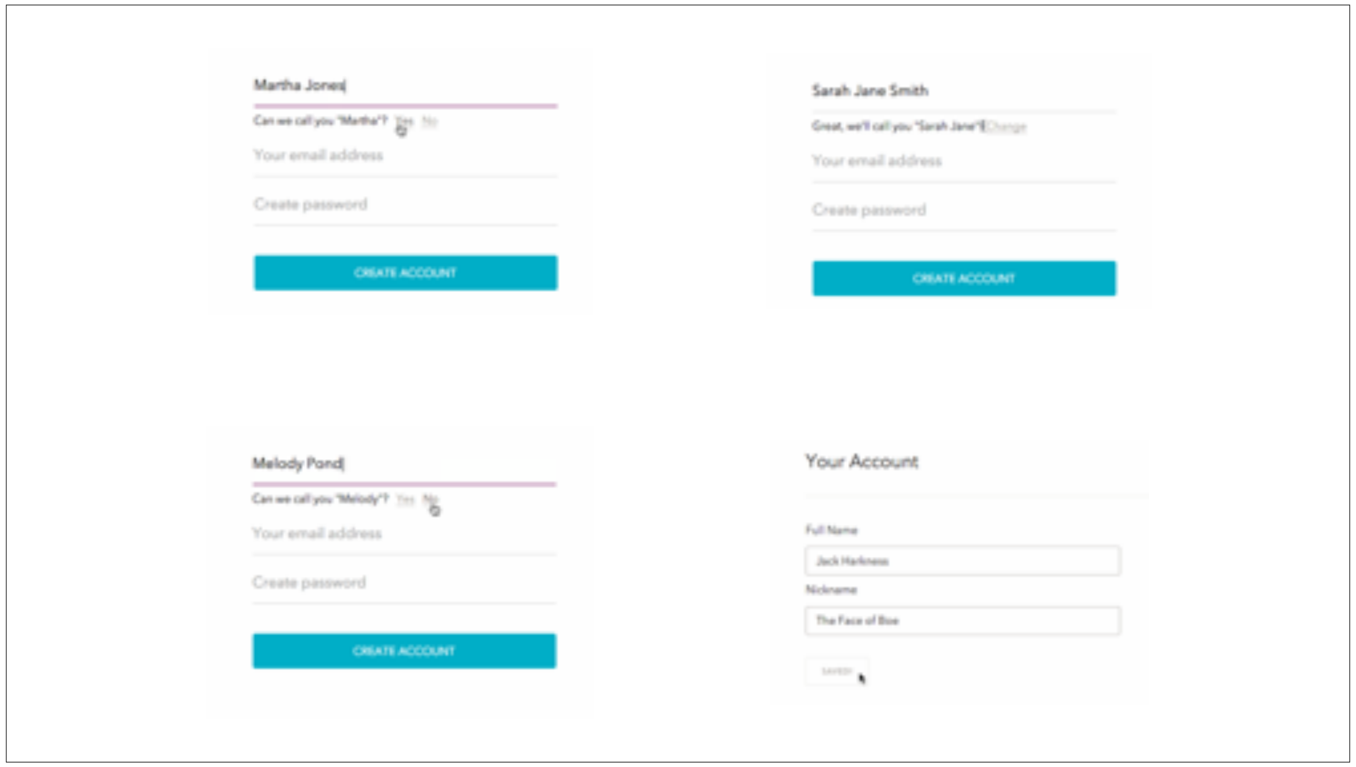
Your Account

Full Name: Jack Harkness

Nickname: The Face of Boe

We ended up with a really nice system where we try to infer the user's nickname, but give them a low-friction way to confirm or deny the nickname. One subtle thing that we did was remember whether the user actually confirmed the nickname or whether it was just inferred. Once a customer has chosen their own nickname or agreed to be called by their first name, we switch this field to true and fine-tune our heuristics to make this choice extra sticky.

We collectively spent more than a week on implementing this feature, and I loved every minute of it.



Martha Jones

Can we call you "Martha"? Yes No

Your email address

Create password

Sarah Jane Smith

Great, we'll call you "Sarah Jane"! Change

Your email address

Create password

Melody Pond

Can we call you "Melody"? Yes No

Your email address

Create password

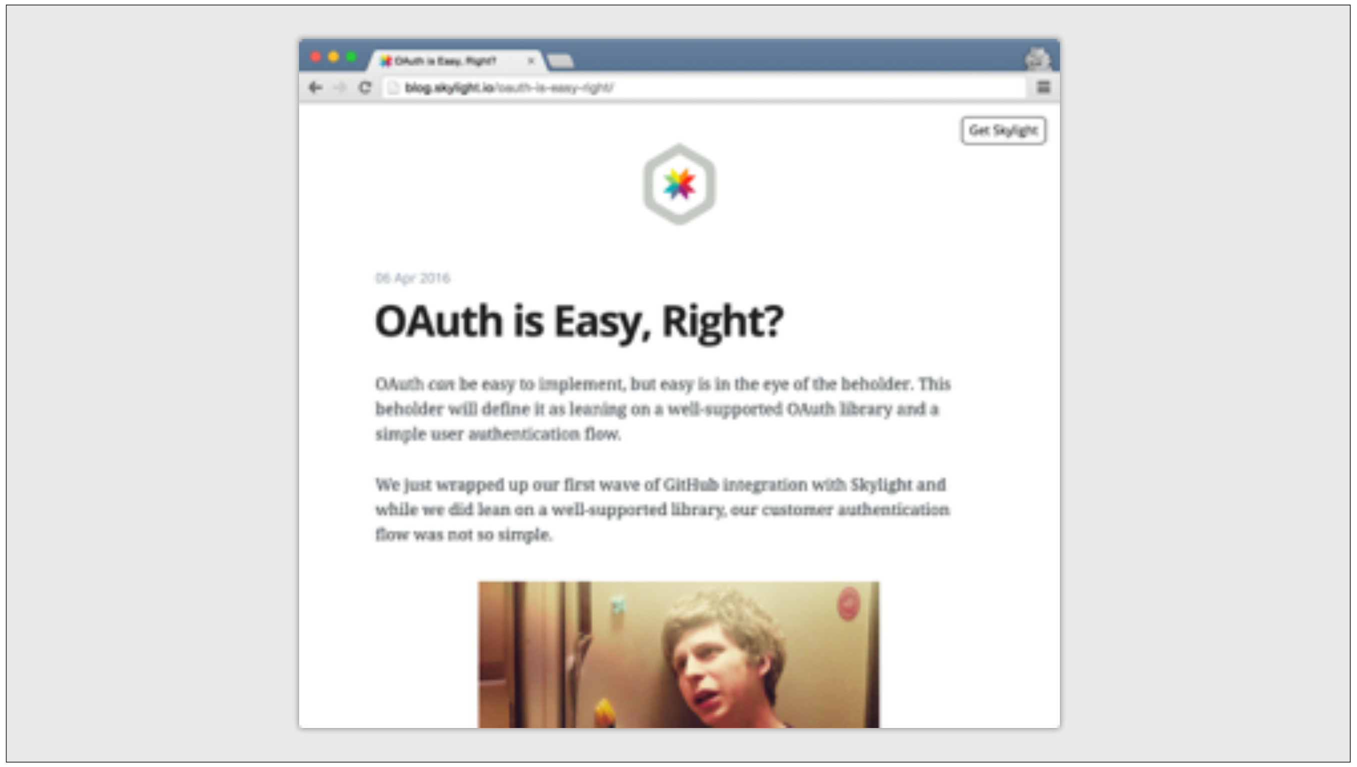
Your Account

Full Name: Jack Harkness

Nickname: The Face of Boe

We ended up with a really nice system where we try to infer the user's nickname, but give them a low-friction way to confirm or deny the nickname. One subtle thing that we did was remember whether the user actually confirmed the nickname or whether it was just inferred. Once a customer has chosen their own nickname or agreed to be called by their first name, we switch this field to true and fine-tune our heuristics to make this choice extra sticky.

We collectively spent more than a week on implementing this feature, and I loved every minute of it.



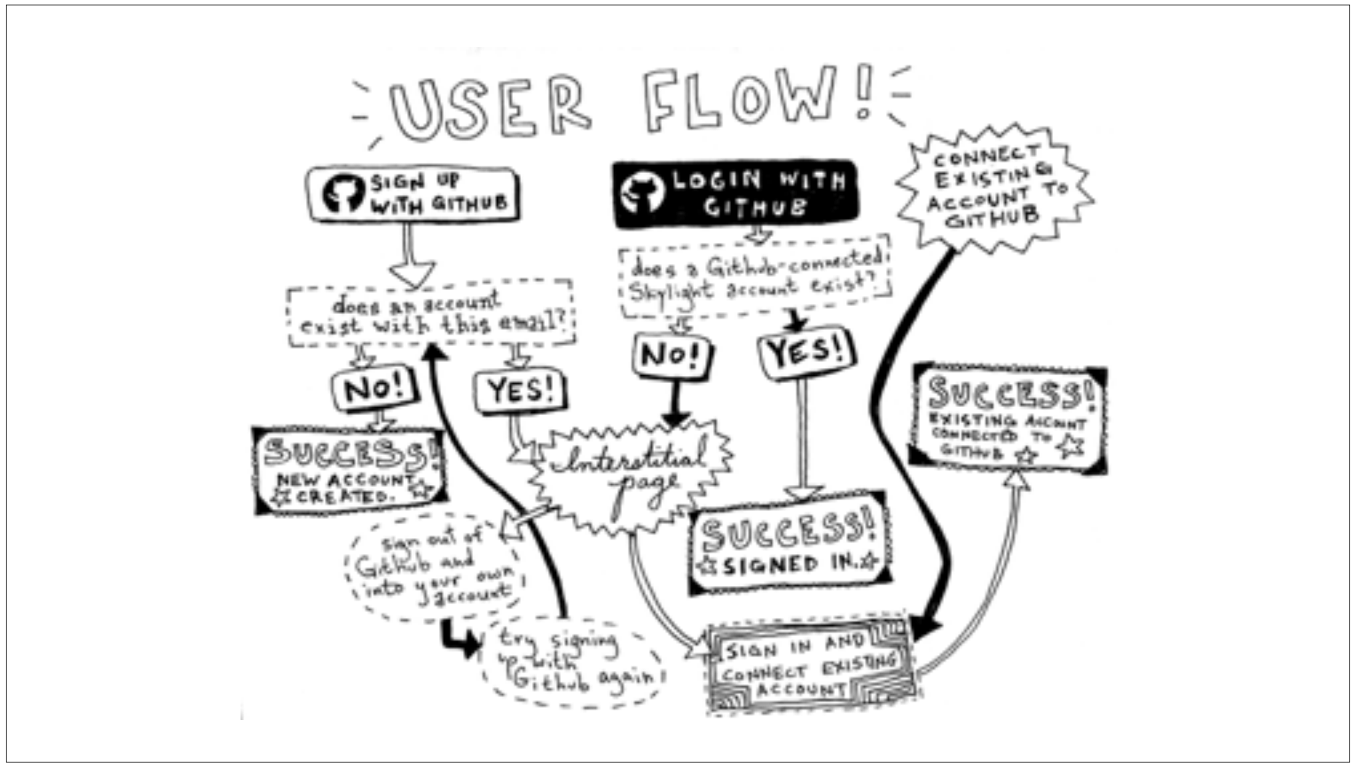
06 Apr 2016

OAuth is Easy, Right?

OAuth *can* be easy to implement, but easy is in the eye of the beholder. This beholder will define it as leaning on a well-supported OAuth library and a simple user authentication flow.

We just wrapped up our first wave of GitHub integration with Skylight and while we did lean on a well-supported library, our customer authentication flow was not so simple.

Another example: when we decided to implement GitHub authentication, we were reminded of many of the ways OAuth signin can be frustrating.

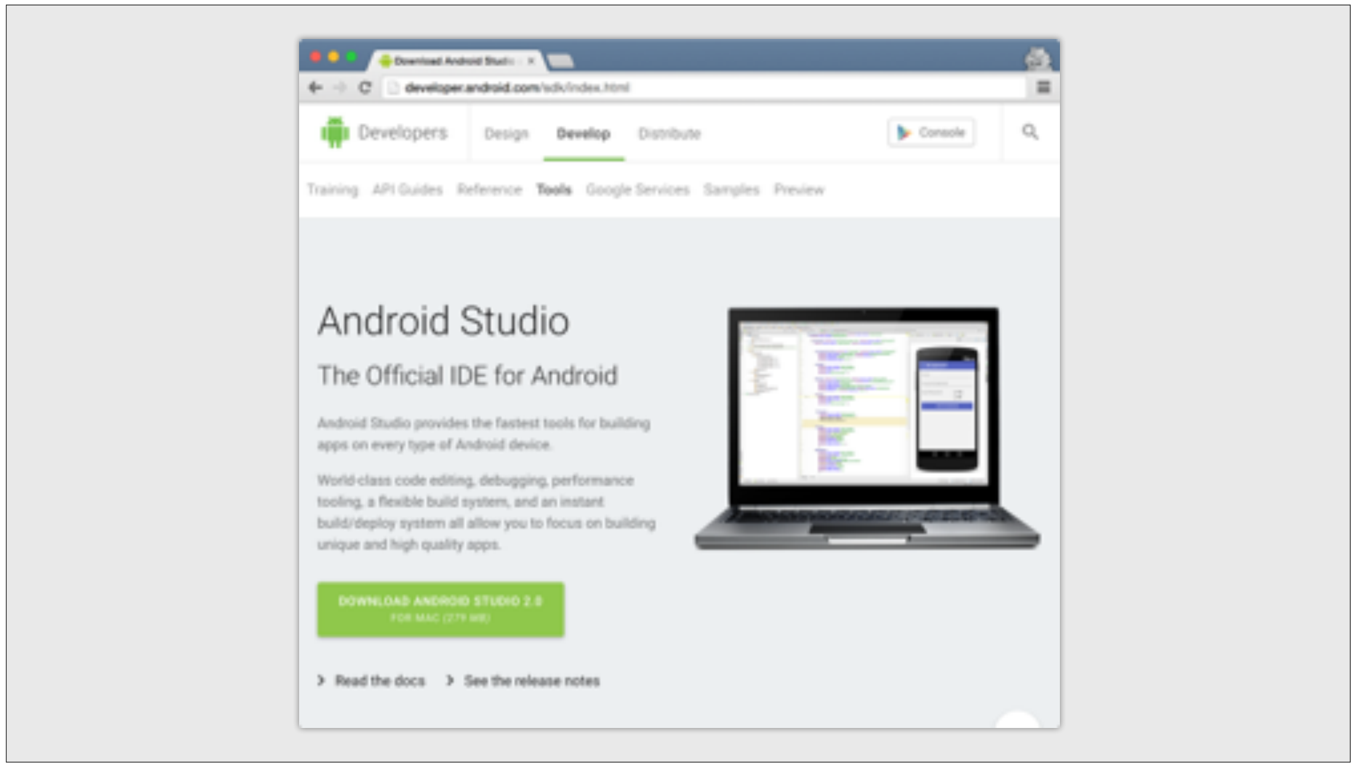

USER FLOW!
SIGN UP WITH GITHUB
LOGIN WITH GITHUB
CONNECT EXISTING ACCOUNT TO GITHUB
does an account exist with this email?
does a Github-connected Skylight account exist?
No!
YES!
No!
YES!
SUCCESS! NEW ACCOUNT CREATED.
Interstitial page
SUCCESS! EXISTING ACCOUNT CONNECTED TO GITHUB
SUCCESS! SIGNED IN.
Sign out of GitHub and into your own account
try signing up with Github again
SIGN IN AND CONNECT EXISTING ACCOUNT

After a lot of discussion, back and forth and prototyping, we ended up with this user-flow (cartoon by Liz Bailey, one of the Tilde engineers who worked on the feature). The blog post has more details, but as always, it's more involved than it looks.

#alllthelittlethings

When I work on products, I want to be able to spend all of my time on getting these details right. When I'm working on Skylight, I want to innovate in Skylight, not on the next iteration of an in-house build pipeline. That doesn't mean Skylight is stuck with old and busted tech: we use persistent web sockets to make our navigation snappy, d3 and SVG for graphs, and Rust in our agent.

Innovation is not at odds with stability. Our native brethren build very innovative apps, without a huge amount of instability. How does that work?



Android Studio

The Official IDE for Android

Android Studio provides the fastest tools for building apps on every type of Android device.

World-class code editing, debugging, performance tooling, a flexible build system, and an instant build/deploy system all allow you to focus on building unique and high quality apps.

Native application platforms provide SDKs, which allow the underlying platform to continue to move forward, maintaining support for existing apps while nudging people towards better patterns and functionality. Android has the Android SDK.

Developing for iOS 9

iOS 9 SDK includes new APIs and services that enable new categories of apps and features. Multitasking and gaming APIs help enhance app functionality and create immersive games. Expanded search capabilities, and new support for CloudKit, HomeKit, HealthKit, and MapKit extend iOS to more places than ever before.

iOS has the iOS SDK. But the web expects users to deal with the primitive changes themselves.

ember

An SDK for the Web

I think of Ember as an SDK for the web, providing a strong level of application compatibility and stability, while working to nudge people towards new capabilities.

Instability is a Drag on Innovation

Fundamentally, instability is a drag on innovation. Native SDKs are quite stable, and people think native apps are more innovative than web apps. There is no paradox: instability slows people and ecosystems down.

You Can Build Higher and Faster on a Stable Foundation

When the foundation isn't shifting wildly, you can innovate in your own apps. And like I said, I love doing that kind of work. And I think we should start thinking of innovation in terms of our own apps. Building amazing experiences is fun, hard, but possible.

We Got This So Wrong

We ourselves fell into the trap of believing the "move fast and break things" mantra.

We thought that if our competitors had a feature we didn't, our users would leave en masse.

In fact, it was our instability that alienated early adopters. We got a bad reputation for it. People criticized us because they felt burned.

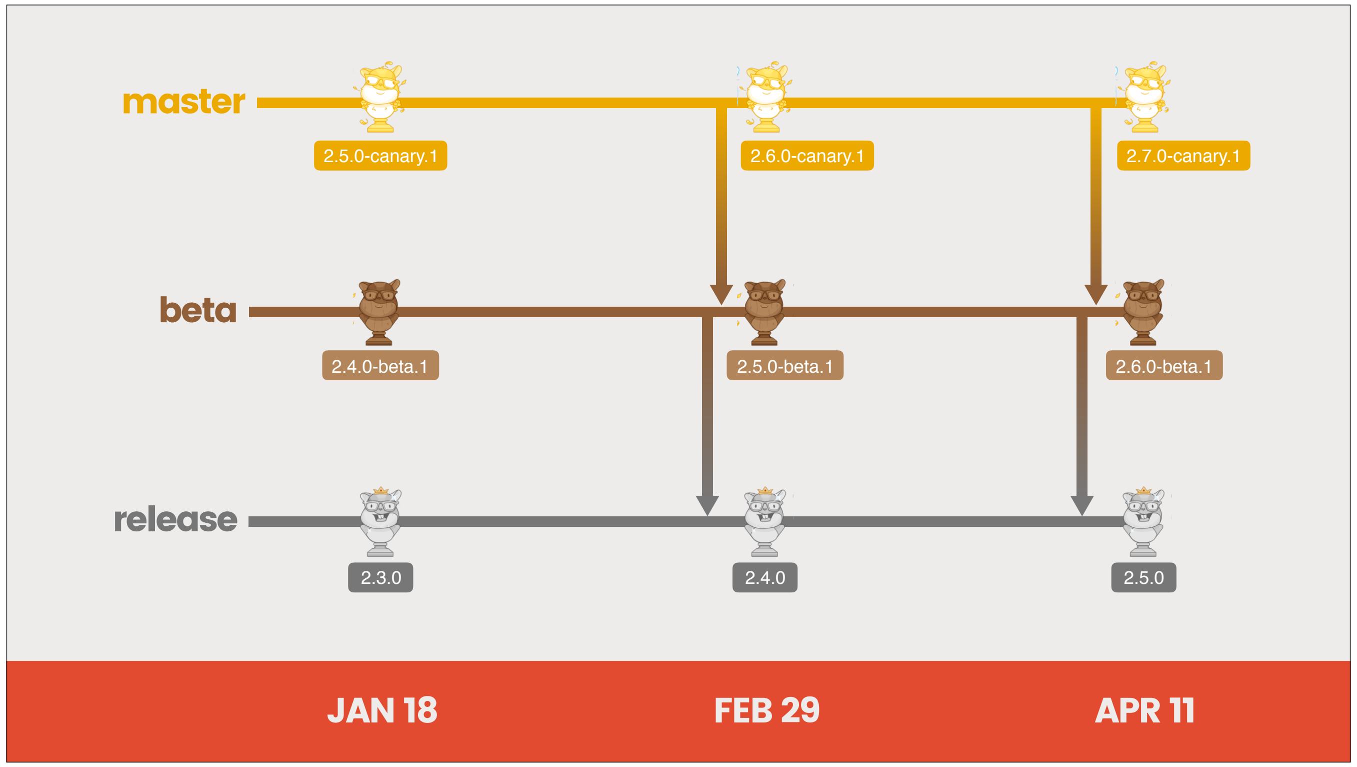
Users don't migrate over night. You have a much larger window than you think (although not infinite).

How do you get the feature in front of new users?

New functionality, like most things, is not a software problem but a distribution problem. How do you get the feature in front of new users?

In order to help deal with the fact that not everyone has the same tolerance for instability, we have release channels, which we'll get into more in details.

* Canary gets the new features as soon as they pass tests, but those features can change or even be removed.
* Beta gets the features once we're pretty sure they're ready, but there's still a chance the features have bugs, might change in minor ways, or in extreme cases be removed.
* Release gets the features once they've been through the beta process and we're ready to make semver guarantees.



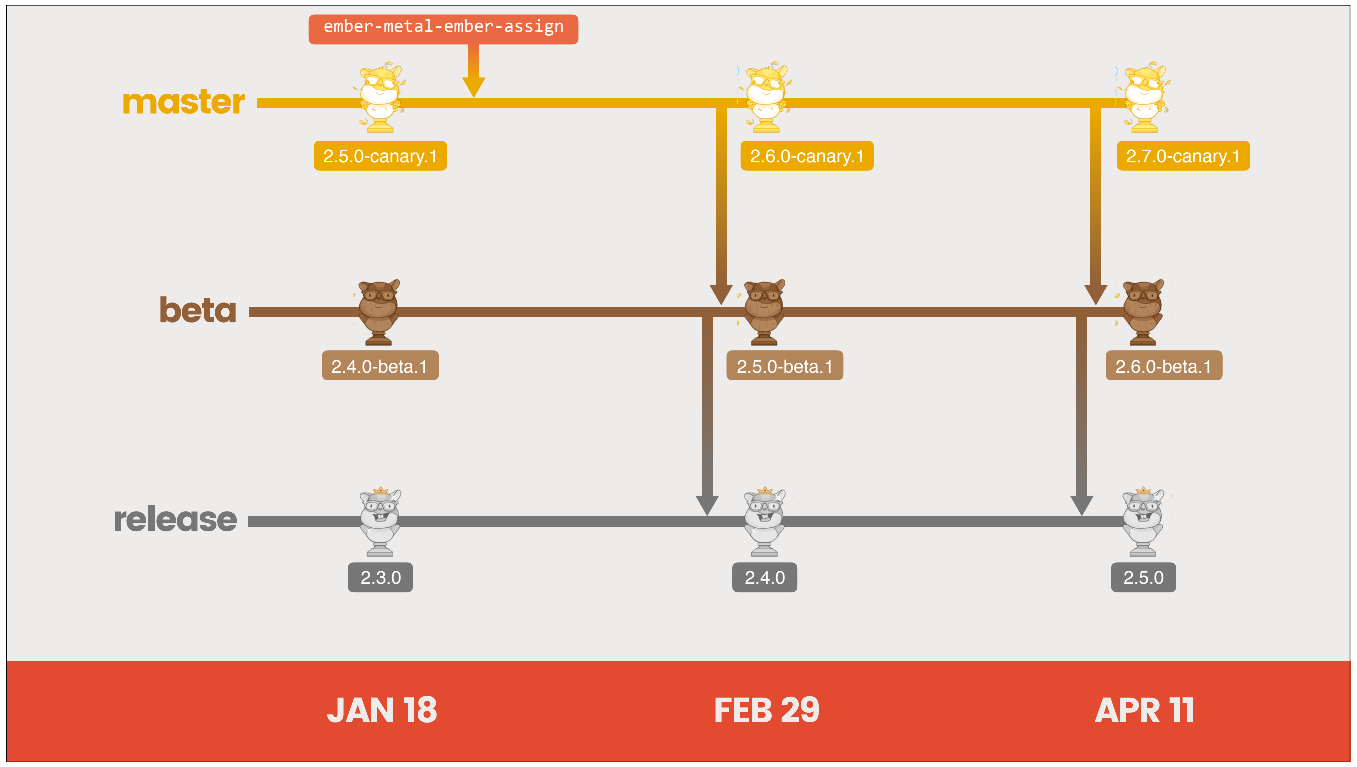
Here's the flow of the process. If you're familiar with the Chrome process, it's pretty much a direct clone.

Lifecycle of a Simple Feature

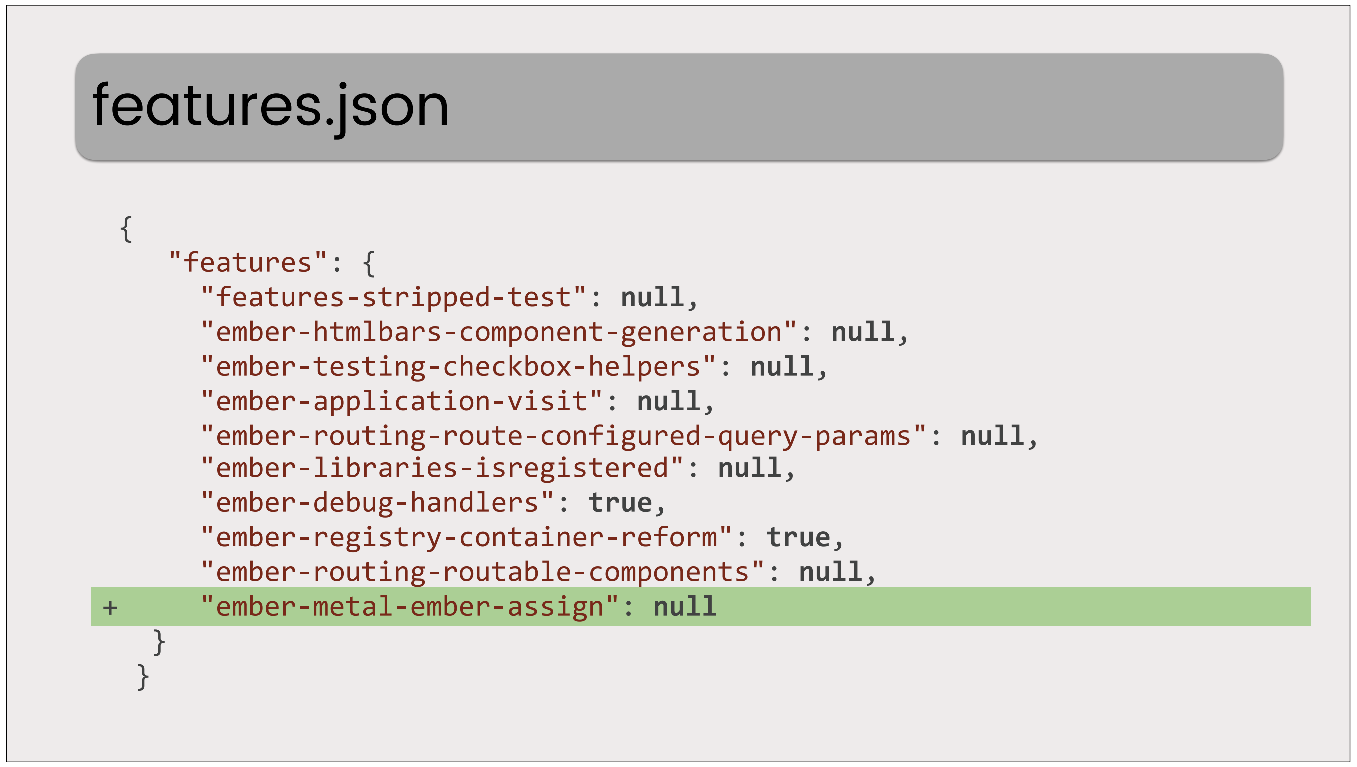
But that's pretty abstract. To understand what's going on, let's take a look at what the process means for a very simple feature.

ember-metal-ember-assign

This feature made it into Ember 2.5. How did it get there?



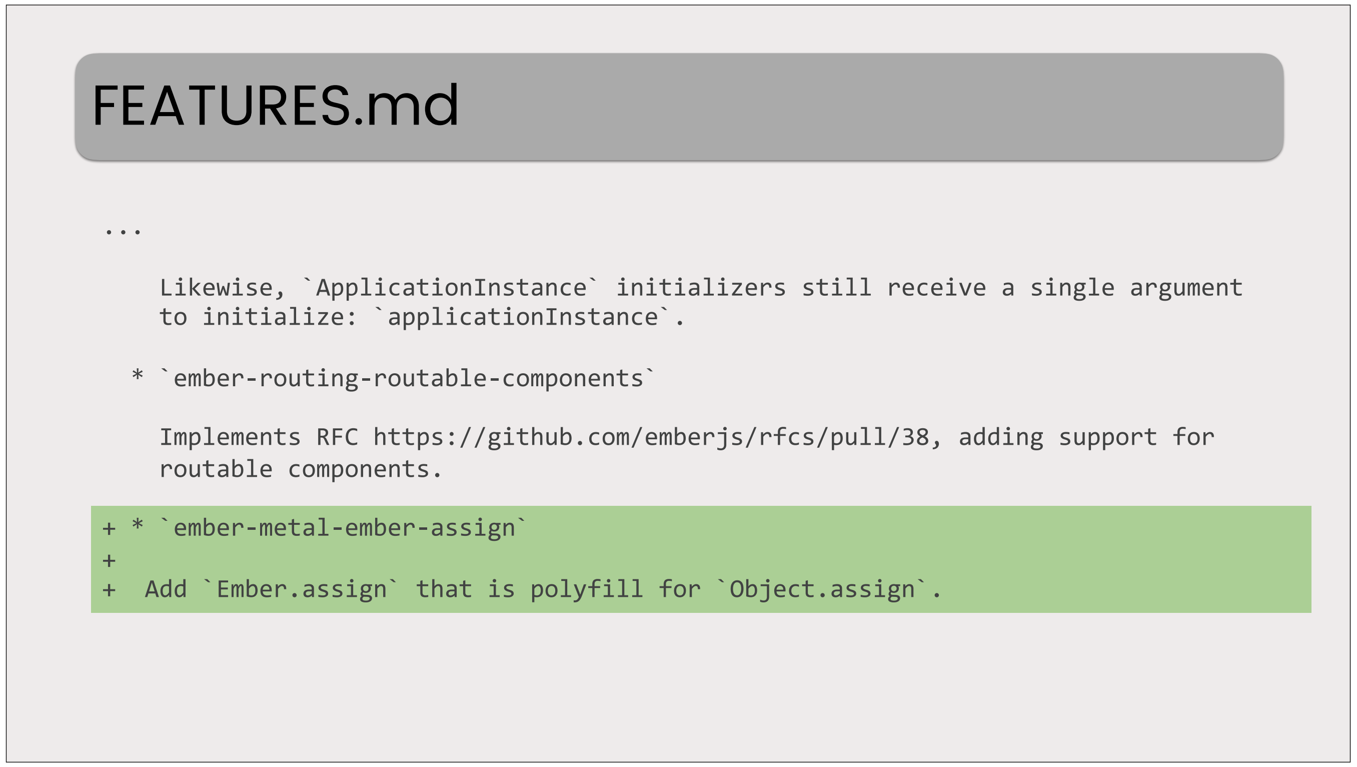
First of all, it landed on the master branch during the 2.5 canary period.



features.json

```
{
   "features": {
     "features-stripped-test": null,
     "ember-htmlbars-component-generation": null,
     "ember-testing-checkbox-helpers": null,
     "ember-application-visit": null,
     "ember-routing-route-configured-query-params": null,
     "ember-libraries-isregistered": null,
     "ember-debug-handlers": true,
     "ember-registry-container-reform": true,
     "ember-routing-routable-components": null,
+    "ember-metal-ember-assign": null
   }
 }
```

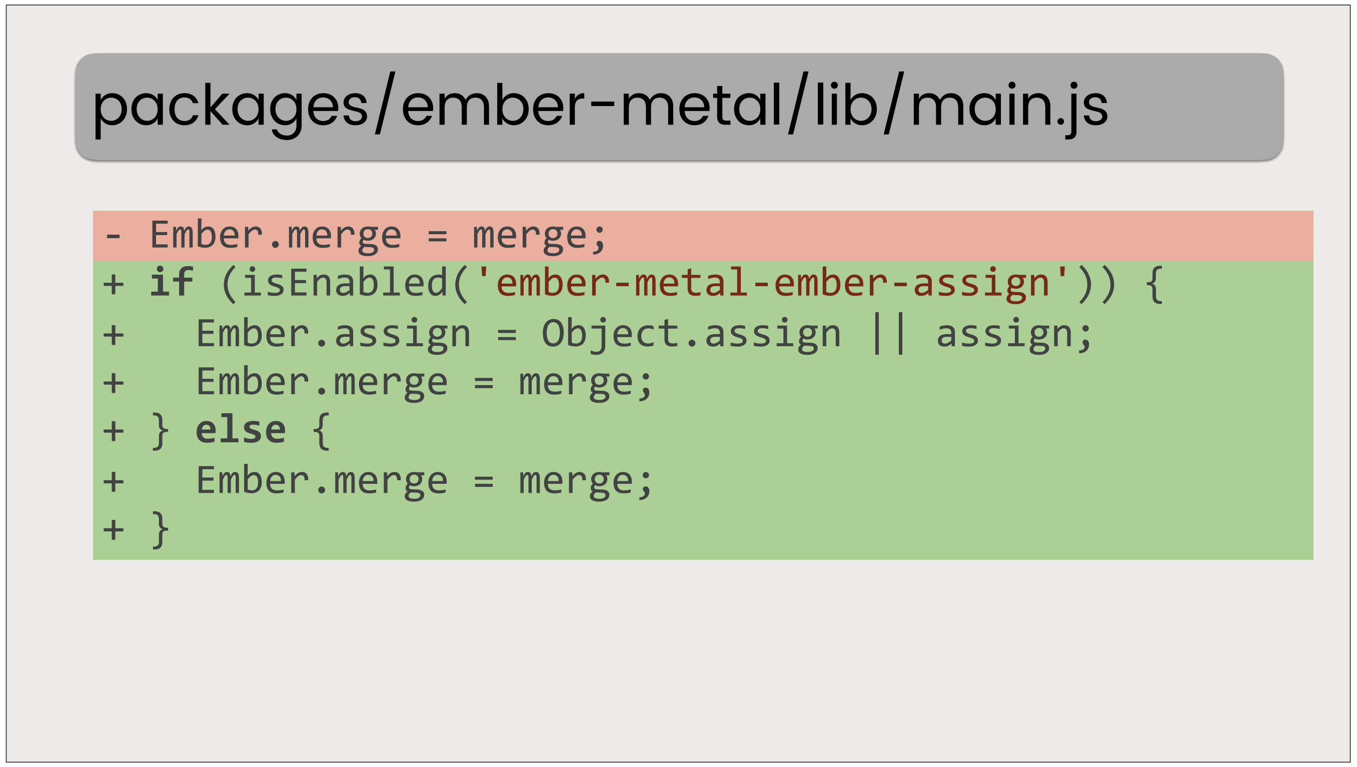
The features.json is updated to reflect the new feature. For now, the feature is marked as disabled. This means it will not be included in beta or release builds, and that you must explicitly opt into them in Canary.



FEATURES.md

```
...

    Likewise, `ApplicationInstance` initializers still receive a single argument
    to initialize: `applicationInstance`.

  * `ember-routing-routable-components`

    Implements RFC https://github.com/emberjs/rfcs/pull/38, adding support for
    routable components.

+ * `ember-metal-ember-assign`
+
+  Add `Ember.assign` that is polyfill for `Object.assign`.
```

And an entry is added to FEATURES.md. This helps the community track in-flight features.

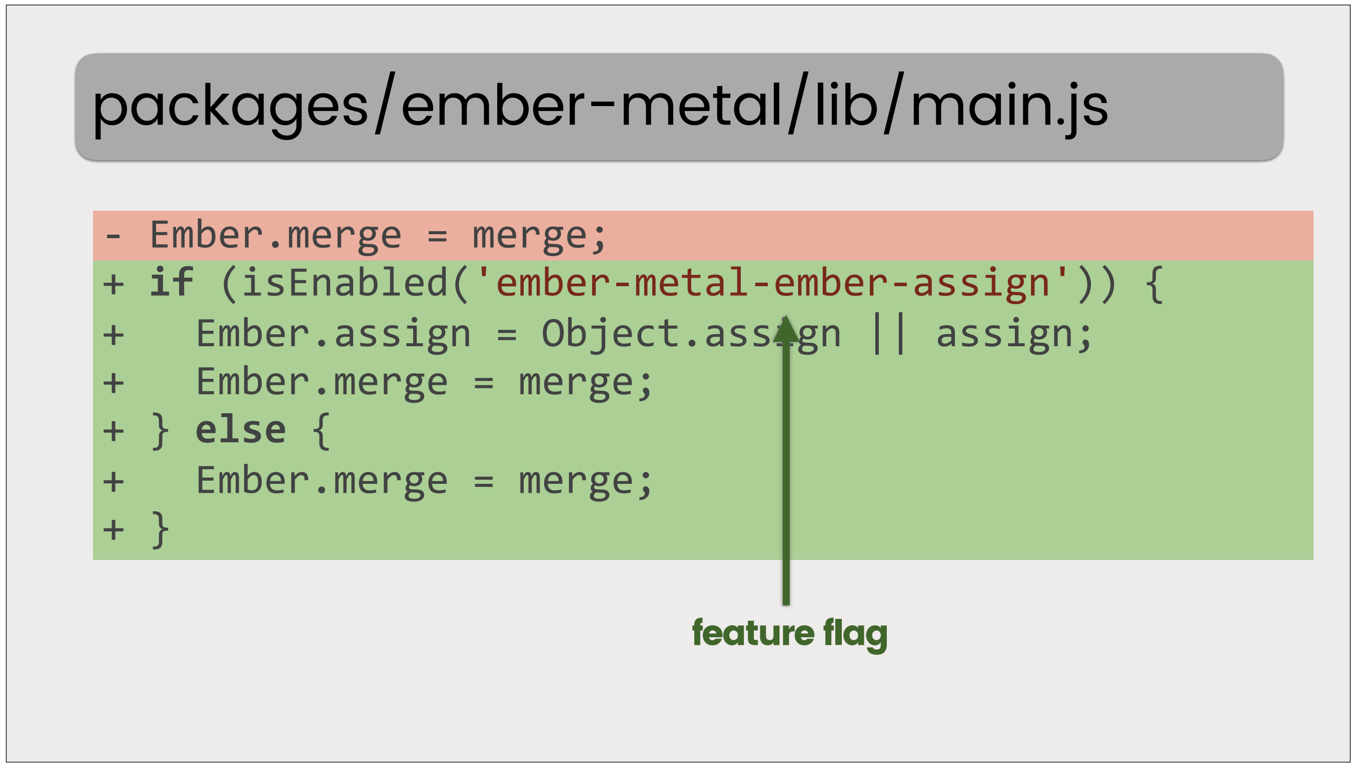


packages/ember-metal/lib/main.js

```
- Ember.merge = merge;
+ if (isEnabled('ember-metal-ember-assign')) {
+   Ember.assign = Object.assign || assign;
+   Ember.merge = merge;
+ } else {
+   Ember.merge = merge;
+ }
```

Then, the feature is implemented. (click) note the feature flag.

All publicly accessible code for a feature that changes the public API **must** be placed behind a feature flag.

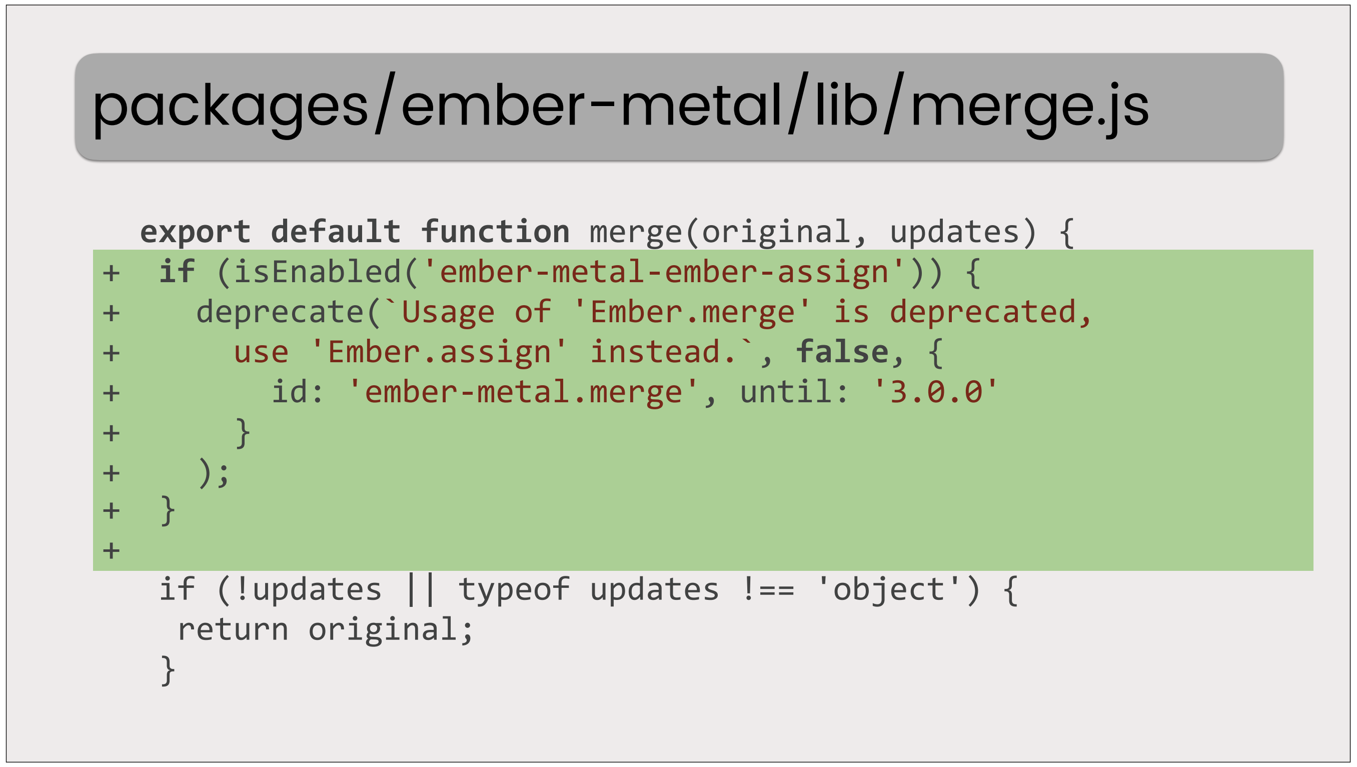


packages/ember-metal/lib/main.js

```
- Ember.merge = merge;
+ if (isEnabled('ember-metal-ember-assign')) {
+   Ember.assign = Object.assign || assign;
+   Ember.merge = merge;
+ } else {
+   Ember.merge = merge;
+ }
```

Then, the feature is implemented. (click) note the feature flag.

All publicly accessible code for a feature that changes the public API **must** be placed behind a feature flag.

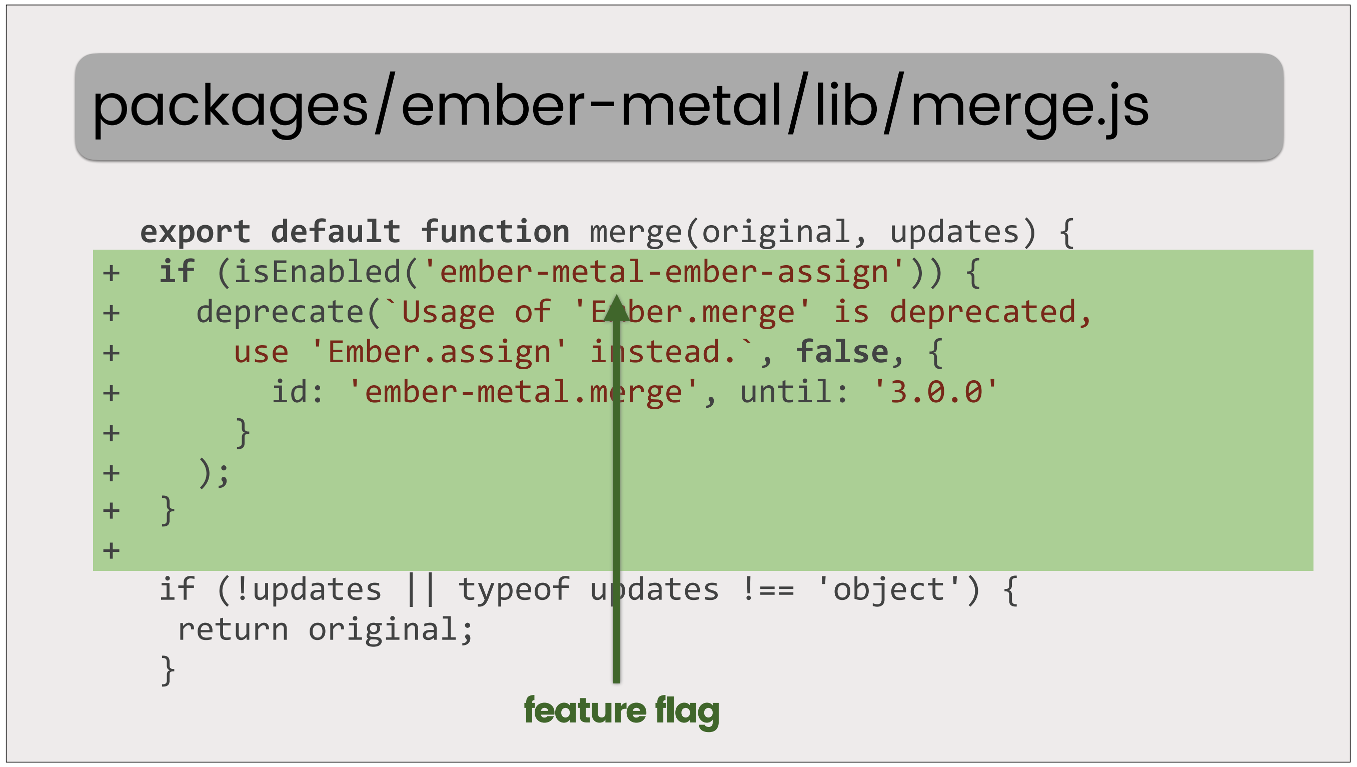


packages/ember-metal/lib/merge.js

```
  export default function merge(original, updates) {
+  if (isEnabled('ember-metal-ember-assign')) {
+    deprecate(`Usage of 'Ember.merge' is deprecated,
+      use 'Ember.assign' instead.`, false, {
+        id: 'ember-metal.merge', until: '3.0.0'
+      }
+    );
+  }
+
   if (!updates || typeof updates !== 'object') {
    return original;
   }
```

In this case, this feature also resulted in the deprecation of an Ember-specific API that is now being replaced by a more precise polyfill. Since it's a breaking change, the previous API will not be removed until Ember 3.0.0. The metadata provided here is used by the Ember inspector and the deprecation workflow.

(click) again, the relevant public API changes are guarded by a feature flag.

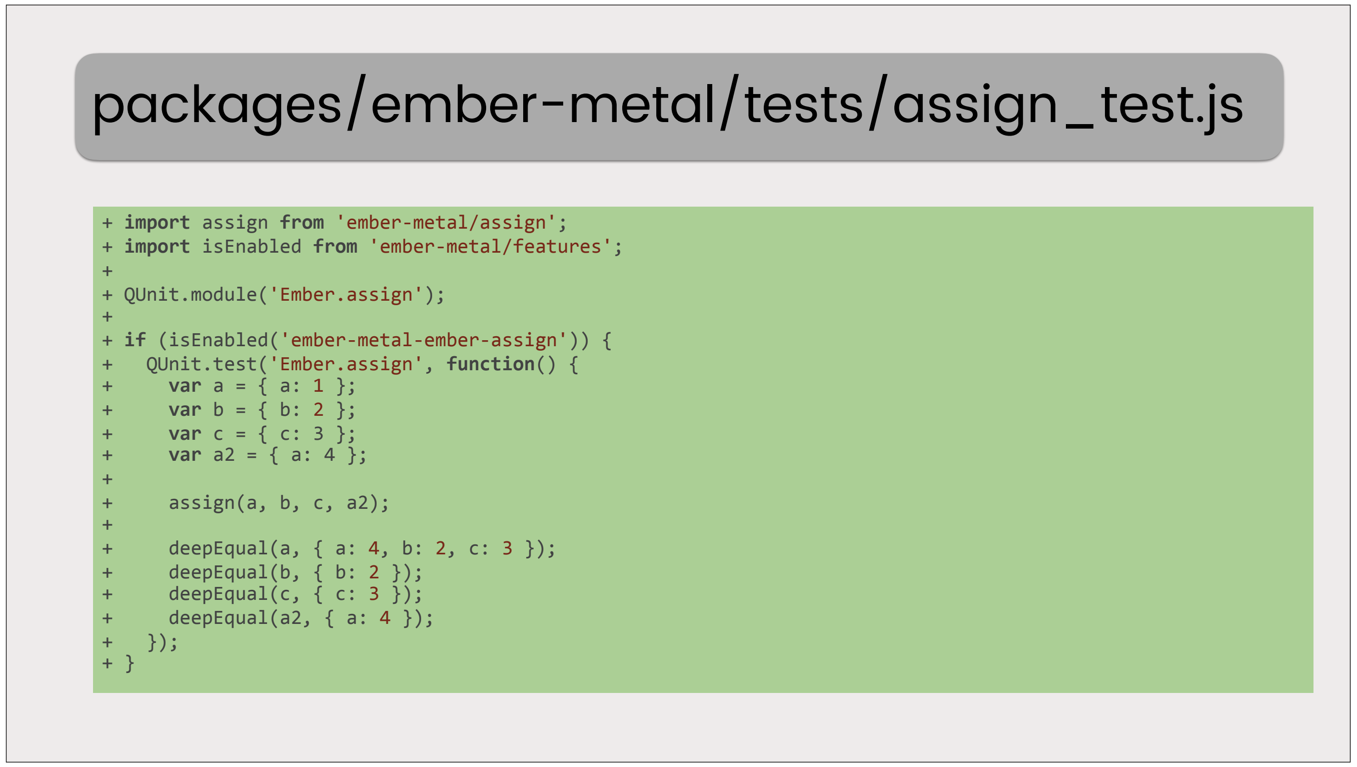


packages/ember-metal/lib/merge.js

```
  export default function merge(original, updates) {
+  if (isEnabled('ember-metal-ember-assign')) {
+    deprecate(`Usage of 'Ember.merge' is deprecated,
+      use 'Ember.assign' instead.`, false, {
+        id: 'ember-metal.merge', until: '3.0.0'
+      }
+    );
+  }
+
   if (!updates || typeof updates !== 'object') {
    return original;
   }
```

In this case, this feature also resulted in the deprecation of an Ember-specific API that is now being replaced by a more precise polyfill. Since it's a breaking change, the previous API will not be removed until Ember 3.0.0. The metadata provided here is used by the Ember inspector and the deprecation workflow.

(click) again, the relevant public API changes are guarded by a feature flag.

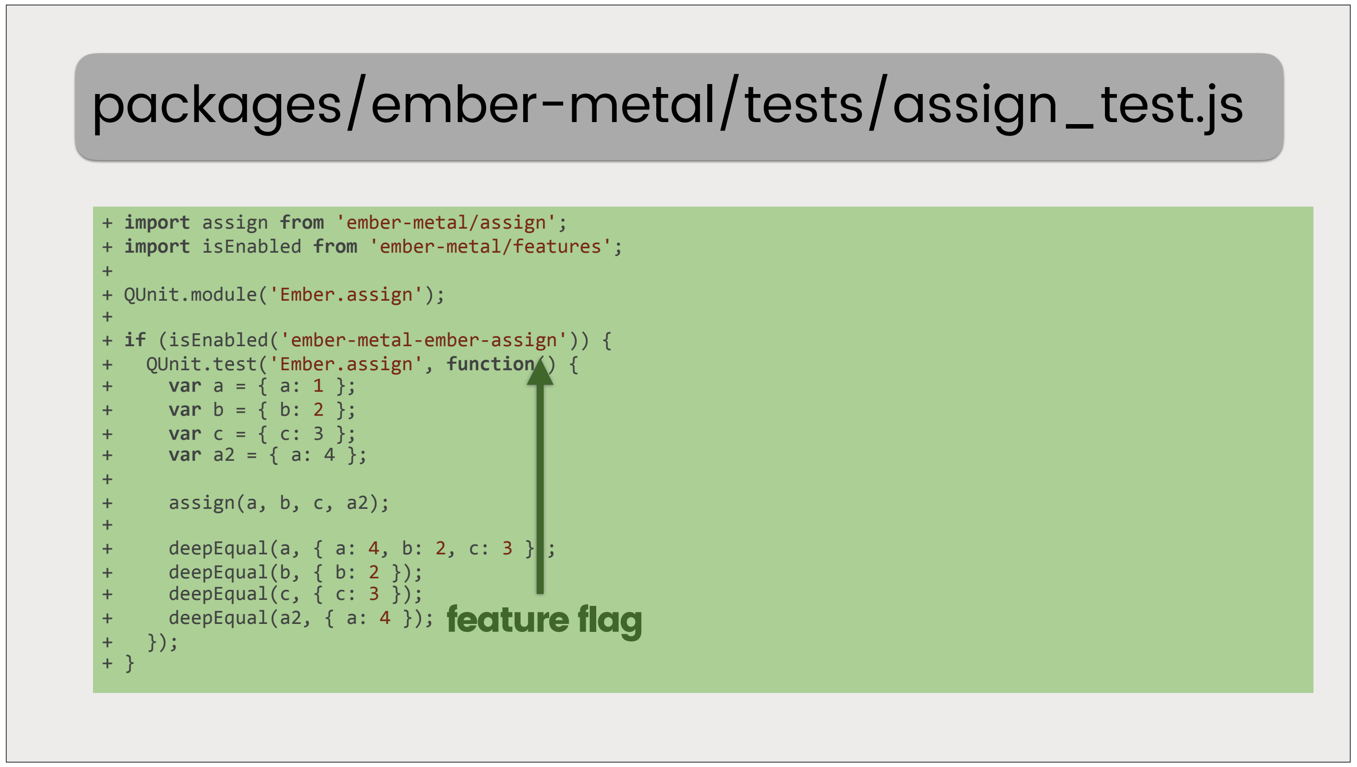


packages/ember-metal/tests/assign_test.js

```
+ import assign from 'ember-metal/assign';
+ import isEnabled from 'ember-metal/features';
+
+ QUnit.module('Ember.assign');
+
+ if (isEnabled('ember-metal-ember-assign')) {
+   QUnit.test('Ember.assign', function() {
+     var a = { a: 1 };
+     var b = { b: 2 };
+     var c = { c: 3 };
+     var a2 = { a: 4 };
+
+     assign(a, b, c, a2);
+
+     deepEqual(a, { a: 4, b: 2, c: 3 });
+     deepEqual(b, { b: 2 });
+     deepEqual(c, { c: 3 });
+     deepEqual(a2, { a: 4 });
+   });
+ }
```

Of course, new features need tests, and tests, too, go behind the feature flag. This allows our CI to run the entire test suite with and without disabled features. One of the great things about this is that it causes us (and PR authors) to notice immediately if they have broken in-progress features. In other words, it keeps all contributors on the same page about work in progress, but we can take our time until the feature is ready before shipping it.

(click) again, the new functionality is wrapped in a feature flag.

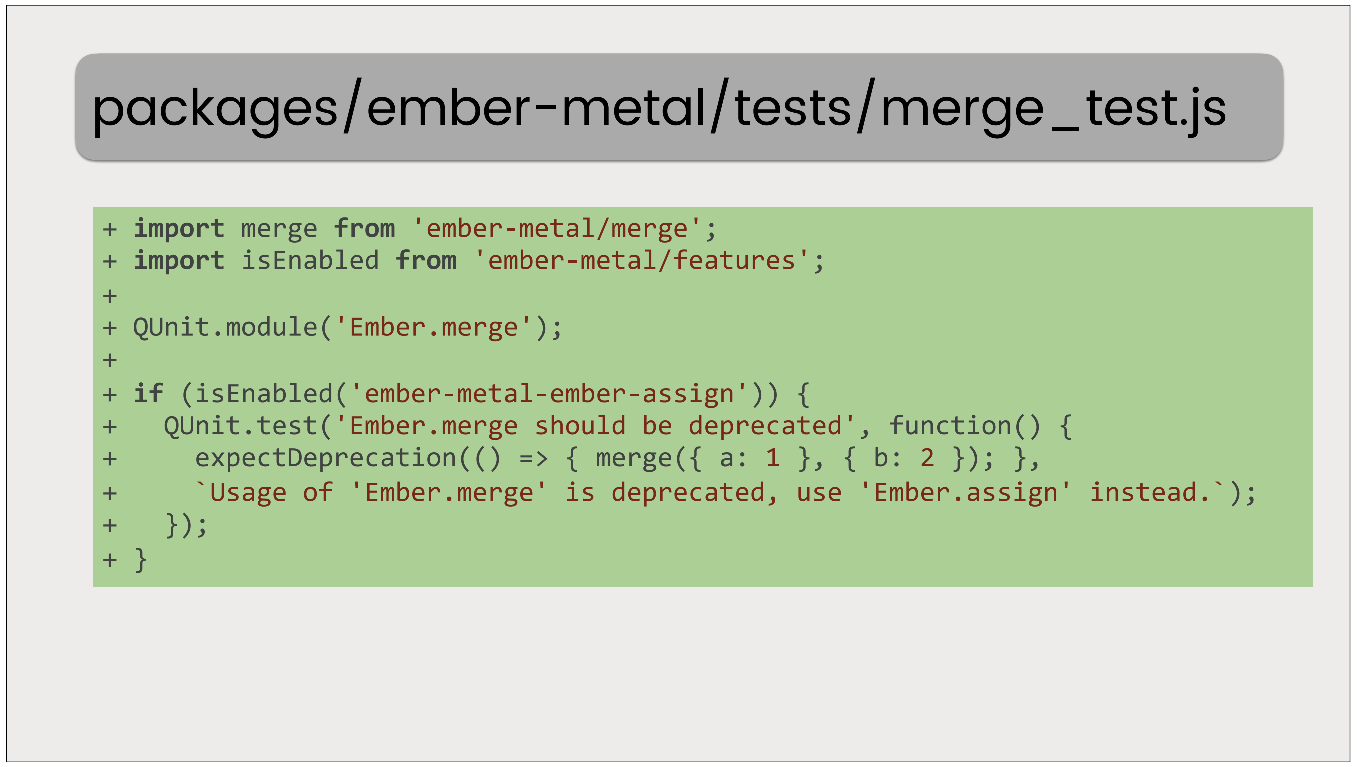


packages/ember-metal/tests/assign_test.js

```
+ import assign from 'ember-metal/assign';
+ import isEnabled from 'ember-metal/features';
+
+ QUnit.module('Ember.assign');
+
+ if (isEnabled('ember-metal-ember-assign')) {
+   QUnit.test('Ember.assign', function() {
+     var a = { a: 1 };
+     var b = { b: 2 };
+     var c = { c: 3 };
+     var a2 = { a: 4 };
+
+     assign(a, b, c, a2);
+
+     deepEqual(a, { a: 4, b: 2, c: 3 });
+     deepEqual(b, { b: 2 });
+     deepEqual(c, { c: 3 });
+     deepEqual(a2, { a: 4 });
+   });
+ }
```

Of course, new features need tests, and tests, too, go behind the feature flag. This allows our CI to run the entire test suite with and without disabled features. One of the great things about this is that it causes us (and PR authors) to notice immediately if they have broken in-progress features. In other words, it keeps all contributors on the same page about work in progress, but we can take our time until the feature is ready before shipping it.

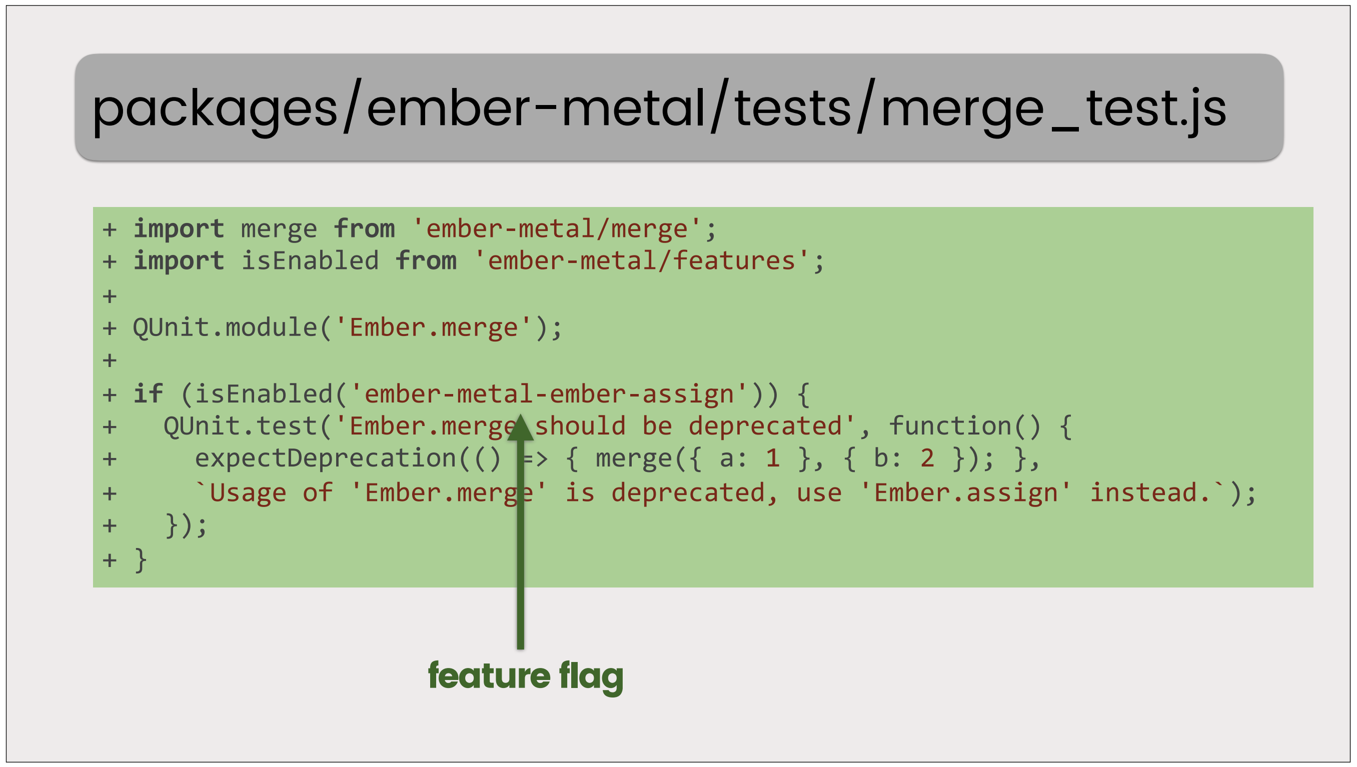
(click) again, the new functionality is wrapped in a feature flag.



packages/ember-metal/tests/merge_test.js

```
+ import merge from 'ember-metal/merge';
+ import isEnabled from 'ember-metal/features';
+
+ QUnit.module('Ember.merge');
+
+ if (isEnabled('ember-metal-ember-assign')) {
+   QUnit.test('Ember.merge should be deprecated', function() {
+     expectDeprecation(() => { merge({ a: 1 }, { b: 2 }); },
+     `Usage of 'Ember.merge' is deprecated, use 'Ember.assign' instead.`);
+   });
+ }
```

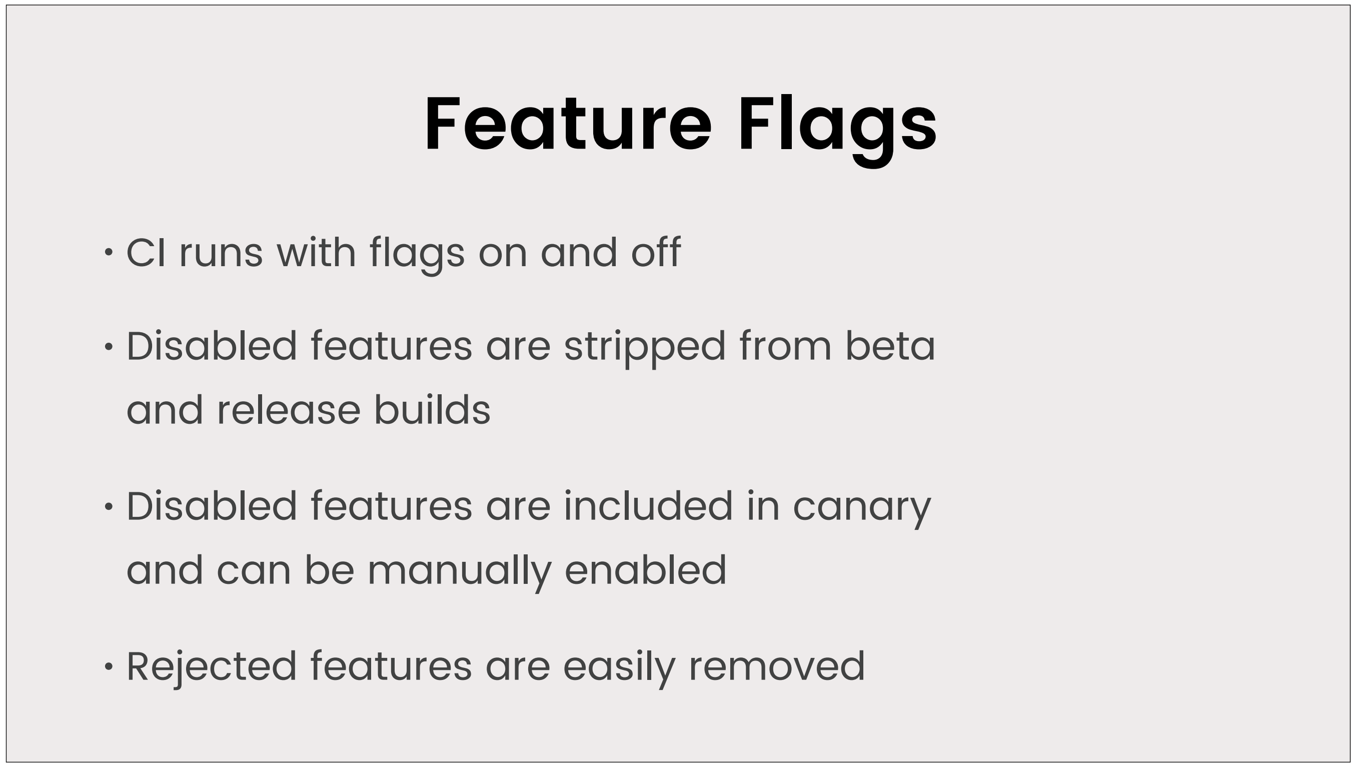
And finally, deprecations are tested, so we don't accidentally regress our deprecation warnings by deleting code or innocent refactors. (click) again, the test is guarded by the same feature flag.



packages/ember-metal/tests/merge_test.js

```
+ import merge from 'ember-metal/merge';
+ import isEnabled from 'ember-metal/features';
+
+ QUnit.module('Ember.merge');
+
+ if (isEnabled('ember-metal-ember-assign')) {
+   QUnit.test('Ember.merge should be deprecated', function() {
+     expectDeprecation(() => { merge({ a: 1 }, { b: 2 }); },
+     `Usage of 'Ember.merge' is deprecated, use 'Ember.assign' instead.`);
+   });
+ }
```

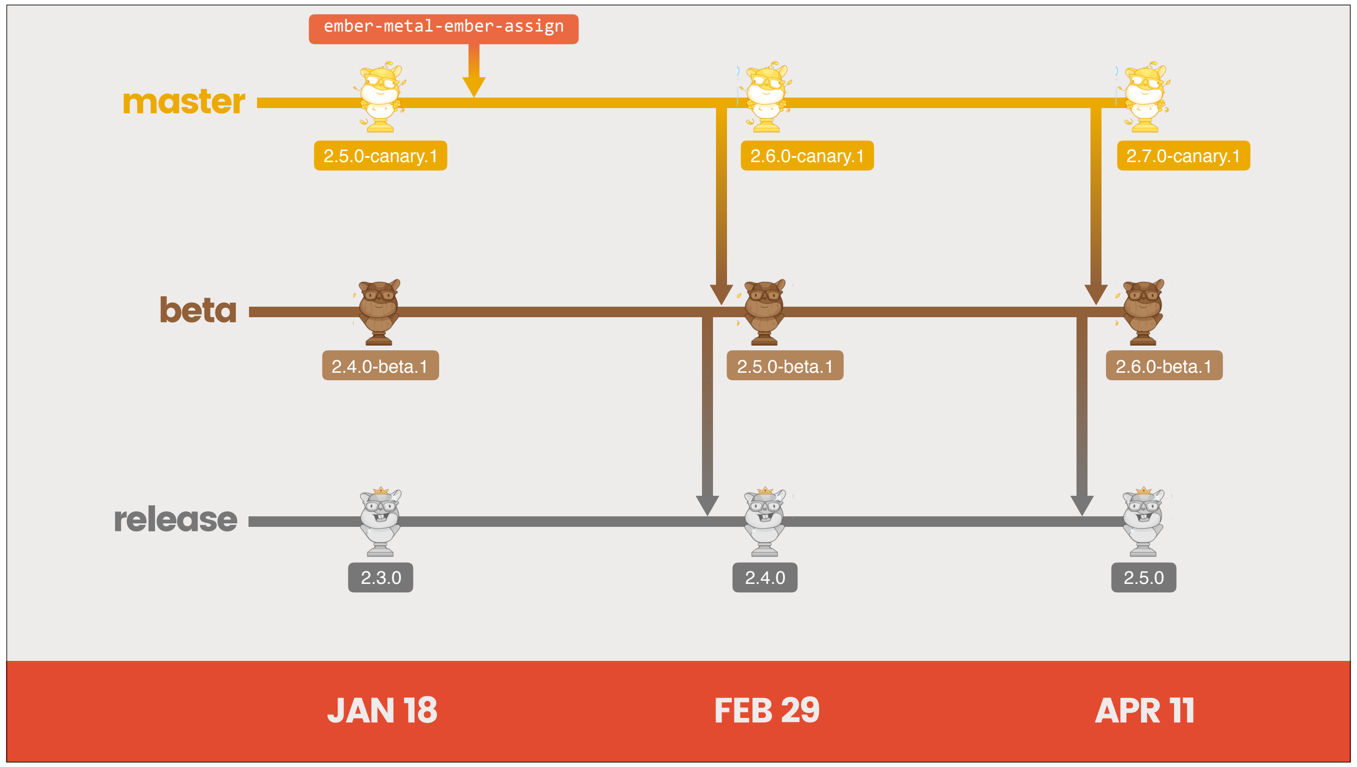
And finally, deprecations are tested, so we don't accidentally regress our deprecation warnings by deleting code or innocent refactors. (click) again, the test is guarded by the same feature flag.



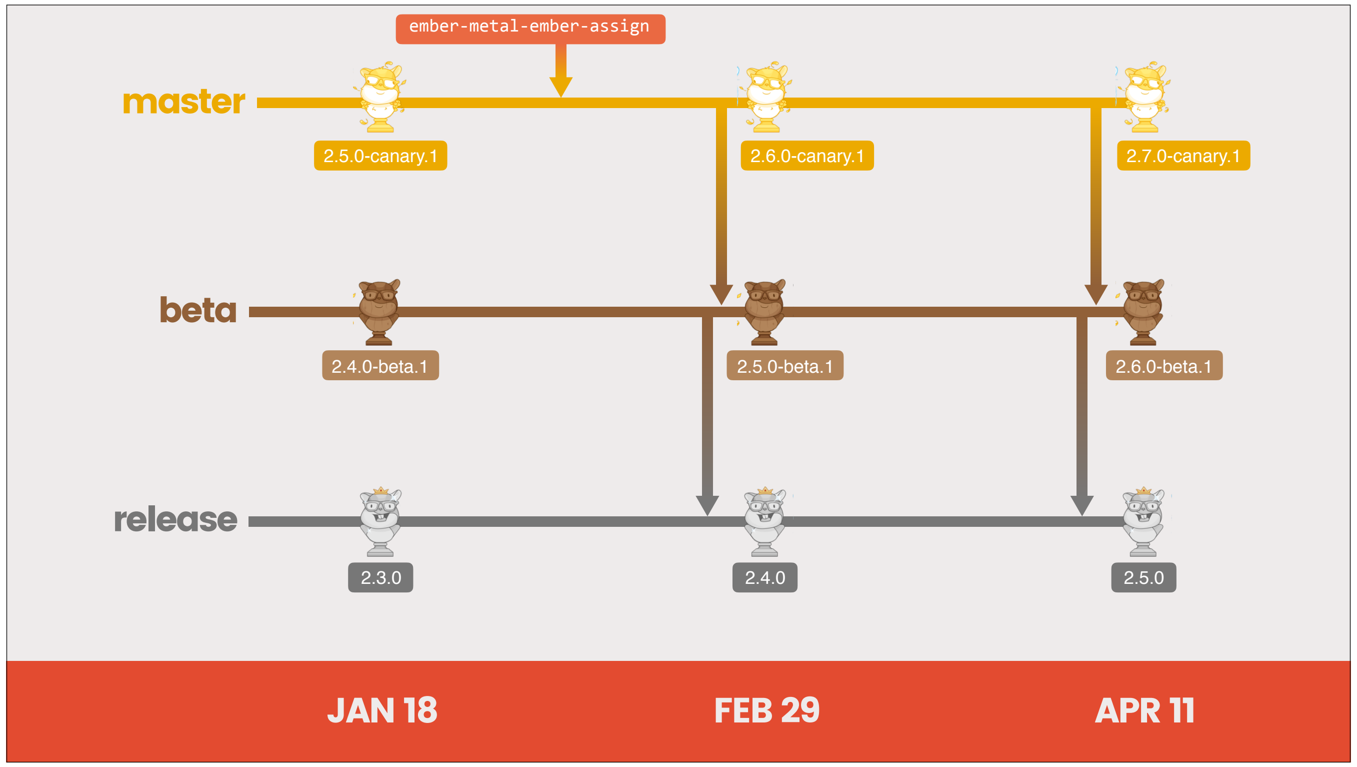
Feature Flags

- CI runs with flags on and off
- Disabled features are stripped from beta and release builds
- Disabled features are included in canary and can be manually enabled
- Rejected features are easily removed

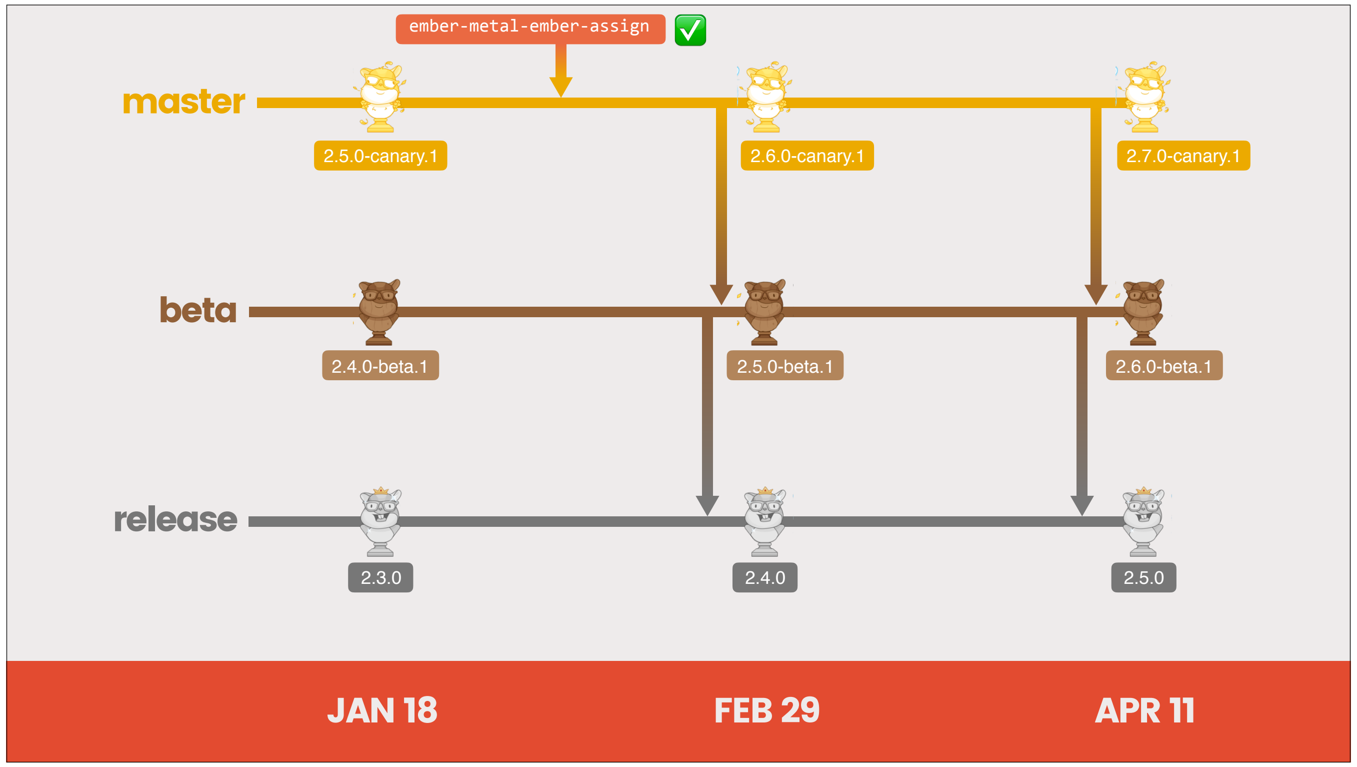
To recap …



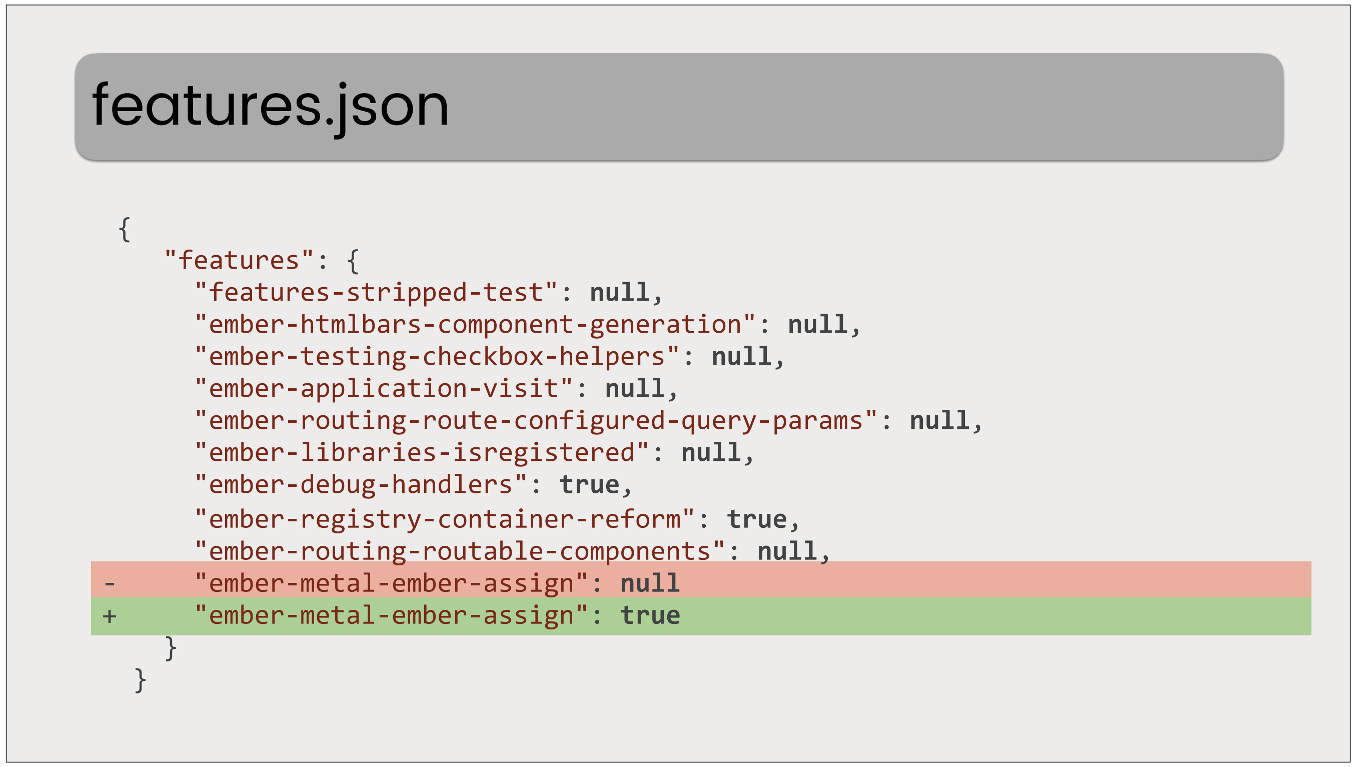
At this point, the feature is available to Canary testers, but it is off by default and won't ship in the next beta. (click) some time passes, and the core team reviews the feature in one of its weekly meetings. (click) after waiting for feedback and reviewing the feature, the core team gives this feature a "go".



At this point, the feature is available to Canary testers, but it is off by default and won't ship in the next beta. (click) some time passes, and the core team reviews the feature in one of its weekly meetings. (click) after waiting for feedback and reviewing the feature, the core team gives this feature a "go".



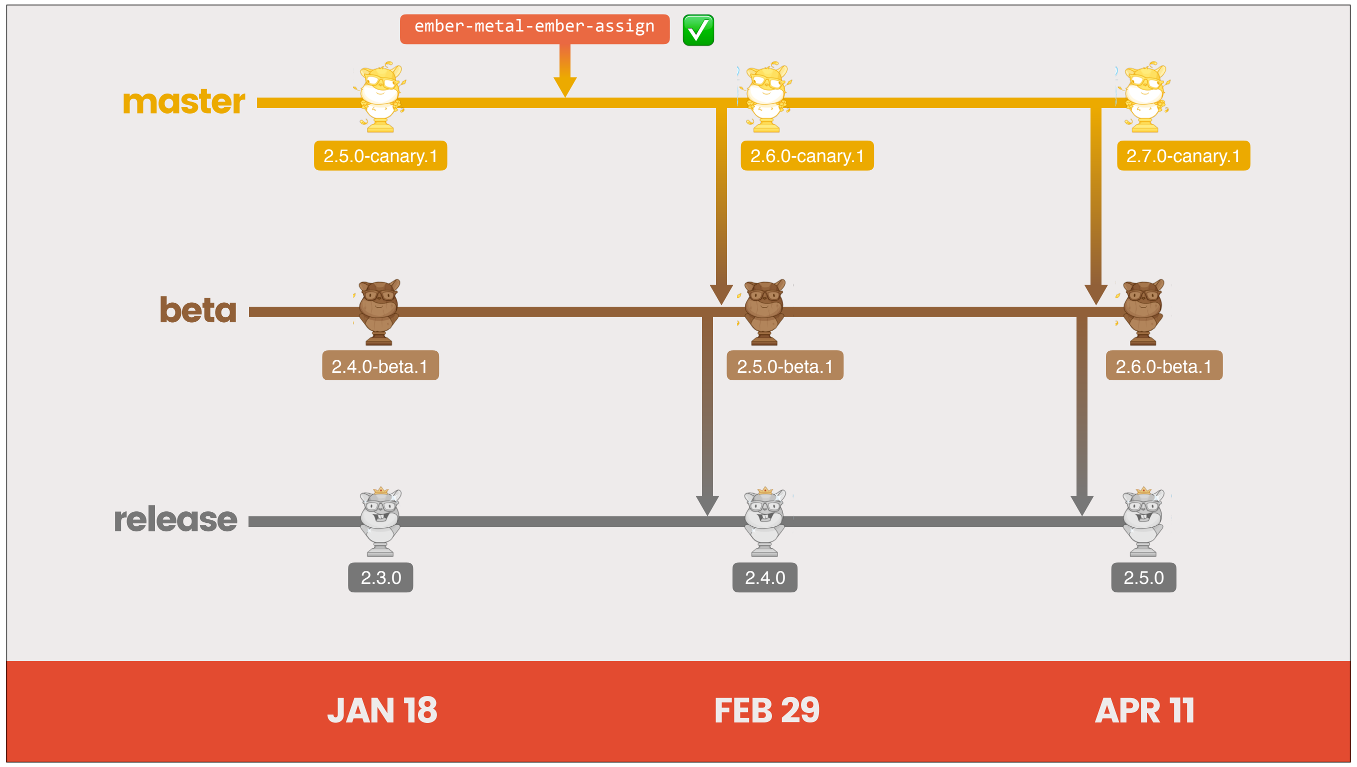
At this point, the feature is available to Canary testers, but it is off by default and won't ship in the next beta. (click) some time passes, and the core team reviews the feature in one of its weekly meetings. (click) after waiting for feedback and reviewing the feature, the core team gives this feature a "go".



features.json

```
{
   "features": {
     "features-stripped-test": null,
     "ember-htmlbars-component-generation": null,
     "ember-testing-checkbox-helpers": null,
     "ember-application-visit": null,
     "ember-routing-route-configured-query-params": null,
     "ember-libraries-isregistered": null,
     "ember-debug-handlers": true,
     "ember-registry-container-reform": true,
     "ember-routing-routable-components": null,
-    "ember-metal-ember-assign": null
+    "ember-metal-ember-assign": true
   }
 }
```

After this happens, we make just one change to the codebase, turning the feature on by default in the features.json. This means that it will be on by default in Canary, and included the next time beta is branched off of master. There is still a chance a problem could be detected in beta, and this gives us the ability to easily roll back the feature if we need to.

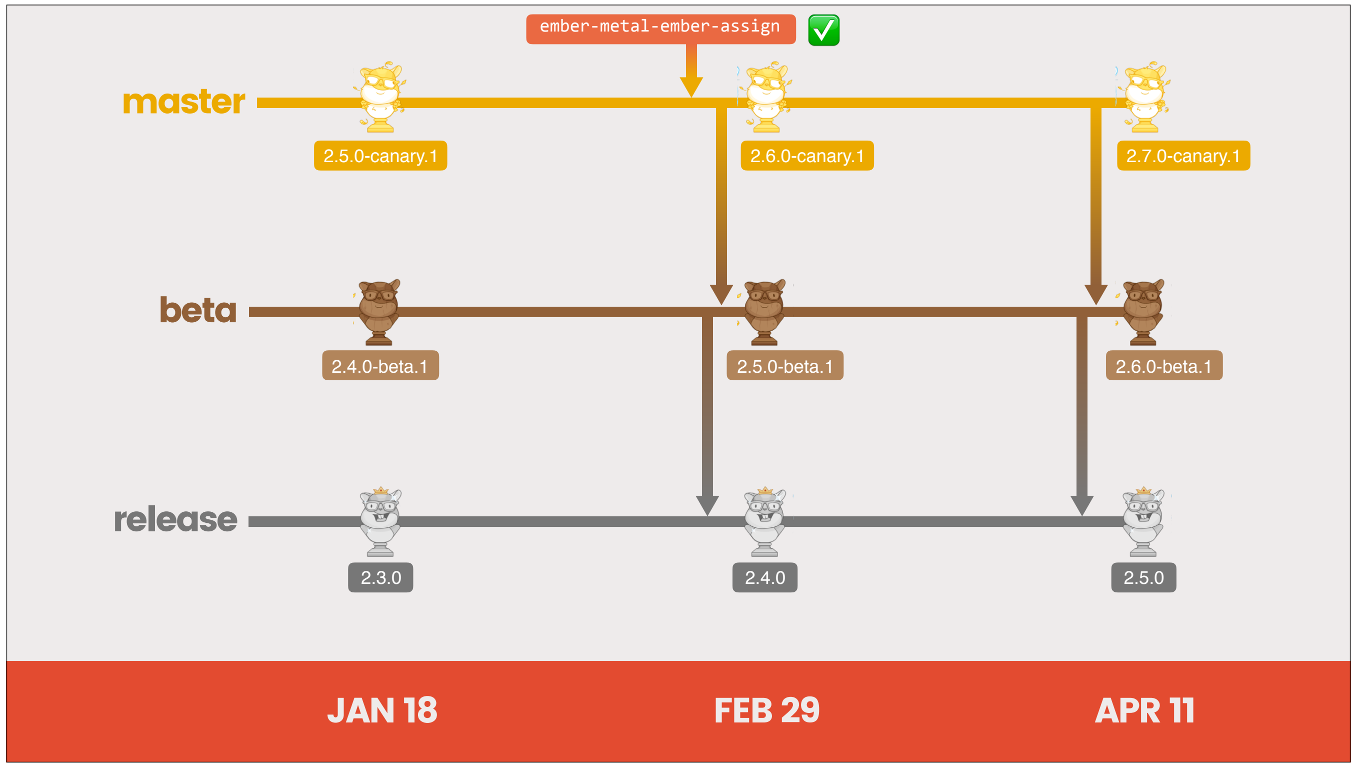


(click) some more time passes, and we're getting ready to branch Ember 2.5 beta from the Ember 2.5 canary branch.

(click) first, we branch the release branch from the previous beta branch.
(click) next, we update the beta branch to master. because the feature is enabled in the features.json, (click) it will be included in the beta build.

(click) now the feature reaches its last test; will it make it through the beta period unscathed? practically speaking, almost every feature that makes it to beta makes it through, but this gives us an opportunity to do one last sanity check with the community. incidentally, the reason features usually make it through is that at this point it's pretty easy to fix any problems that we find during the beta cycle.

(click) some more time passes, and we're ready to release Ember 2.5.
(click) we update the release branch to the latest beta, and since the feature is still enabled, it makes it into the release!

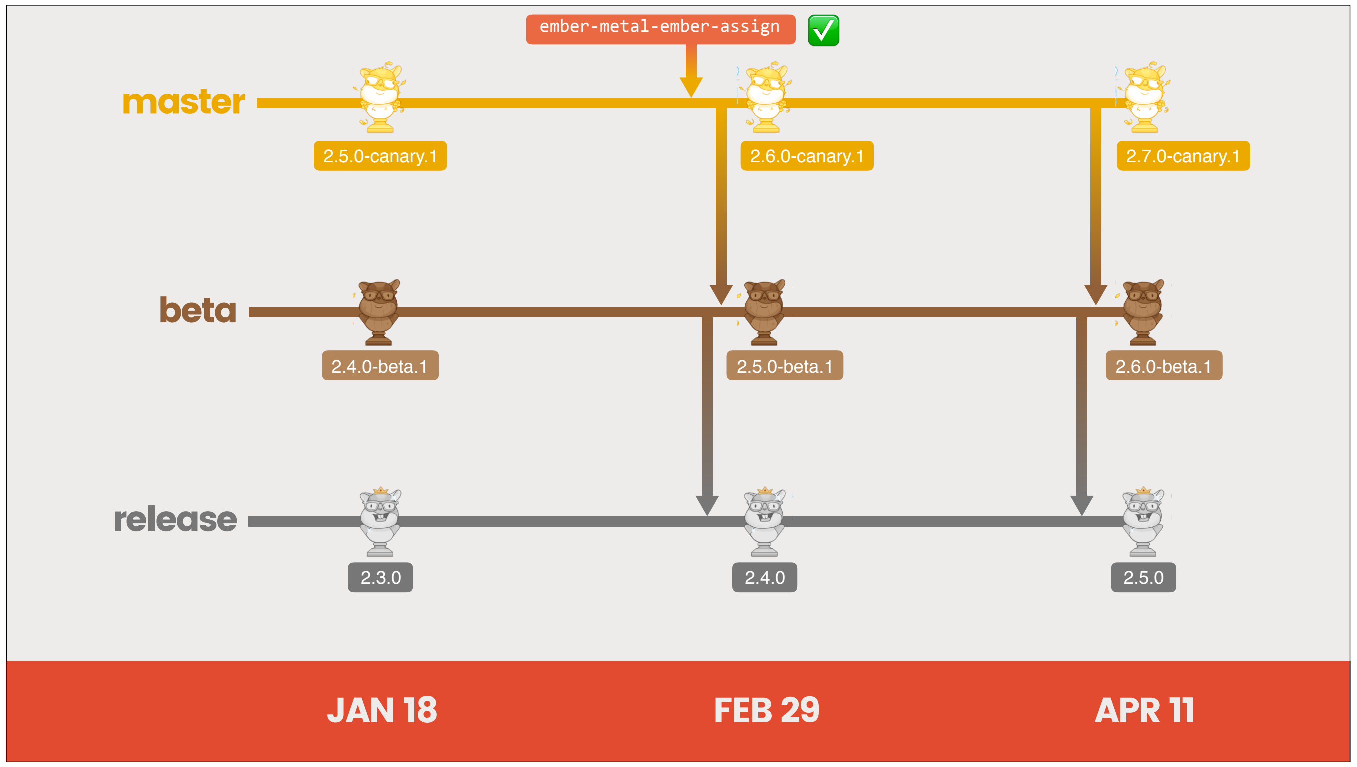


(click) some more time passes, and we're getting ready to branch Ember 2.5 beta from the Ember 2.5 canary branch.

(click) first, we branch the release branch from the previous beta branch.
(click) next, we update the beta branch to master. because the feature is enabled in the features.json, (click) it will be included in the beta build.

(click) now the feature reaches its last test; will it make it through the beta period unscathed? practically speaking, almost every feature that makes it to beta makes it through, but this gives us an opportunity to do one last sanity check with the community. incidentally, the reason features usually make it through is that at this point it's pretty easy to fix any problems that we find during the beta cycle.

(click) some more time passes, and we're ready to release Ember 2.5.
(click) we update the release branch to the latest beta, and since the feature is still enabled, it makes it into the release!

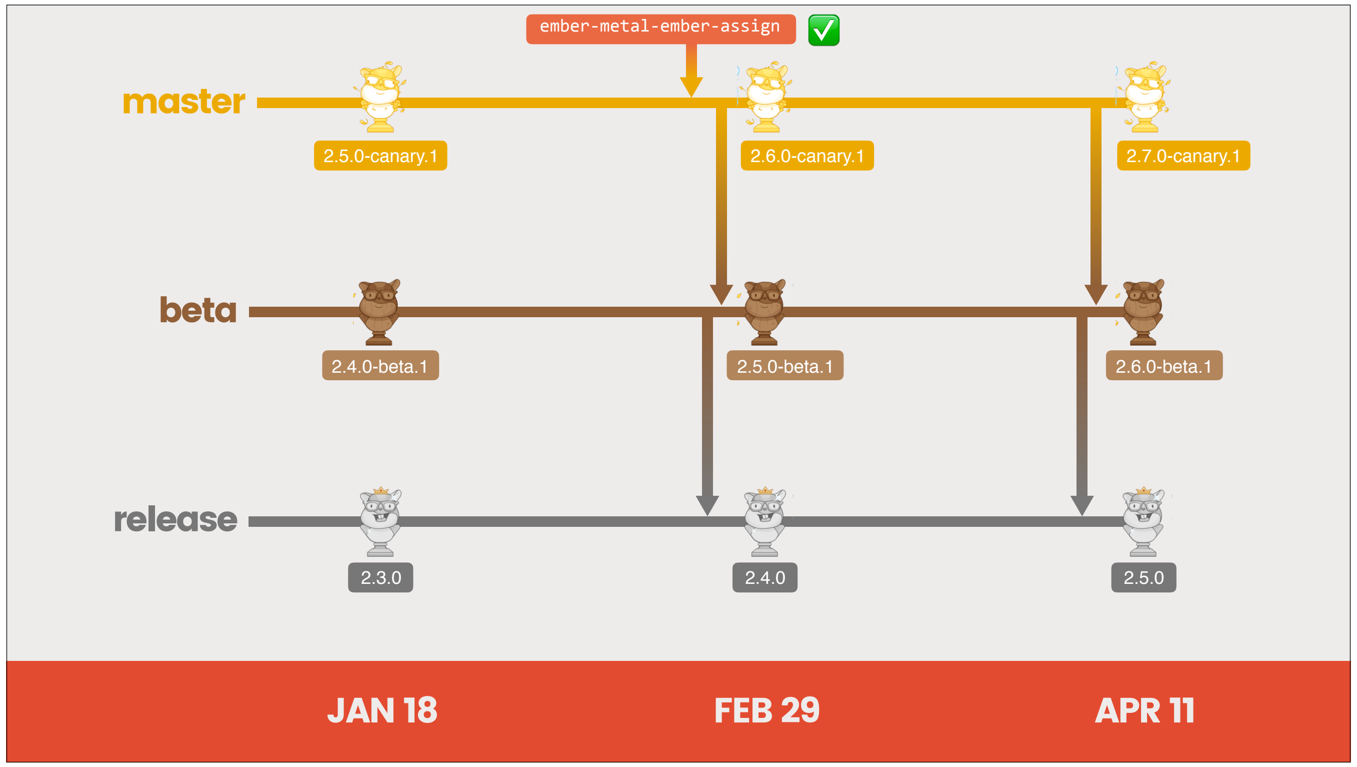


(click) some more time passes, and we're getting ready to branch Ember 2.5 beta from the Ember 2.5 canary branch.

(click) first, we branch the release branch from the previous beta branch.
(click) next, we update the beta branch to master. because the feature is enabled in the features.json, (click) it will be included in the beta build.

(click) now the feature reaches its last test; will it make it through the beta period unscathed? practically speaking, almost every feature that makes it to beta makes it through, but this gives us an opportunity to do one last sanity check with the community. incidentally, the reason features usually make it through is that at this point it's pretty easy to fix any problems that we find during the beta cycle.

(click) some more time passes, and we're ready to release Ember 2.5.
(click) we update the release branch to the latest beta, and since the feature is still enabled, it makes it into the release!

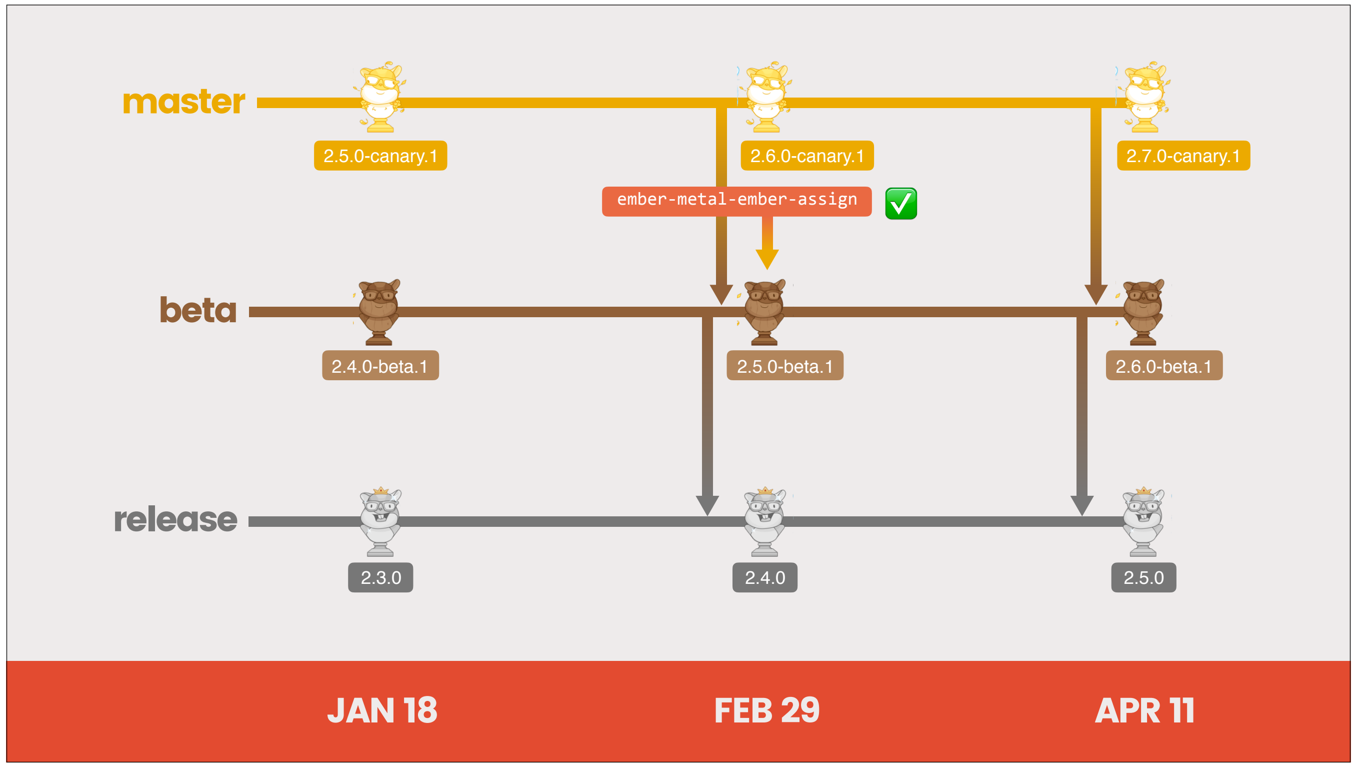


(click) some more time passes, and we're getting ready to branch Ember 2.5 beta from the Ember 2.5 canary branch.

(click) first, we branch the release branch from the previous beta branch.
(click) next, we update the beta branch to master. because the feature is enabled in the features.json, (click) it will be included in the beta build.

(click) now the feature reaches its last test; will it make it through the beta period unscathed? practically speaking, almost every feature that makes it to beta makes it through, but this gives us an opportunity to do one last sanity check with the community. incidentally, the reason features usually make it through is that at this point it's pretty easy to fix any problems that we find during the beta cycle.

(click) some more time passes, and we're ready to release Ember 2.5.
(click) we update the release branch to the latest beta, and since the feature is still enabled, it makes it into the release!

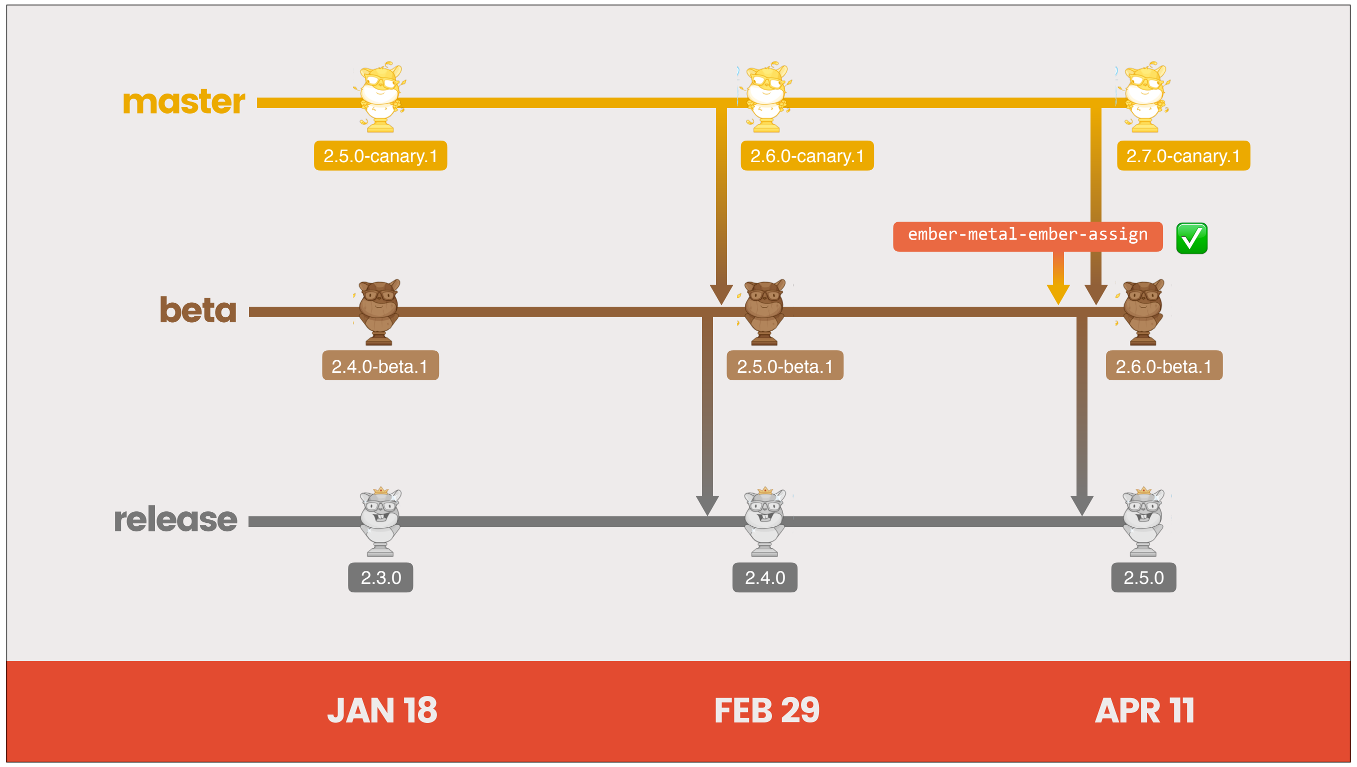


(click) some more time passes, and we're getting ready to branch Ember 2.5 beta from the Ember 2.5 canary branch.

(click) first, we branch the release branch from the previous beta branch.
(click) next, we update the beta branch to master. because the feature is enabled in the features.json, (click) it will be included in the beta build.

(click) now the feature reaches its last test; will it make it through the beta period unscathed? practically speaking, almost every feature that makes it to beta makes it through, but this gives us an opportunity to do one last sanity check with the community. incidentally, the reason features usually make it through is that at this point it's pretty easy to fix any problems that we find during the beta cycle.

(click) some more time passes, and we're ready to release Ember 2.5.
(click) we update the release branch to the latest beta, and since the feature is still enabled, it makes it into the release!

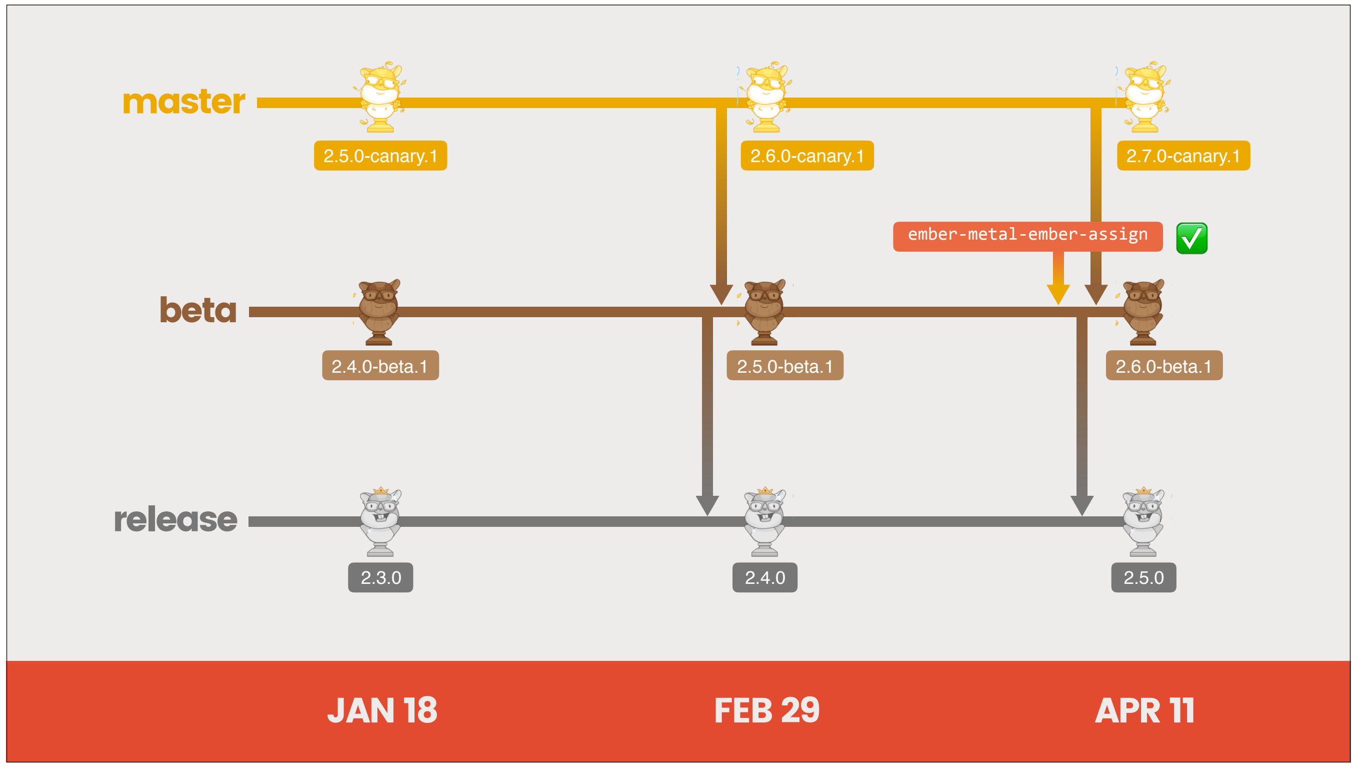


(click) some more time passes, and we're getting ready to branch Ember 2.5 beta from the Ember 2.5 canary branch.

(click) first, we branch the release branch from the previous beta branch.
(click) next, we update the beta branch to master. because the feature is enabled in the features.json, (click) it will be included in the beta build.

(click) now the feature reaches its last test; will it make it through the beta period unscathed? practically speaking, almost every feature that makes it to beta makes it through, but this gives us an opportunity to do one last sanity check with the community. incidentally, the reason features usually make it through is that at this point it's pretty easy to fix any problems that we find during the beta cycle.

(click) some more time passes, and we're ready to release Ember 2.5.
(click) we update the release branch to the latest beta, and since the feature is still enabled, it makes it into the release!

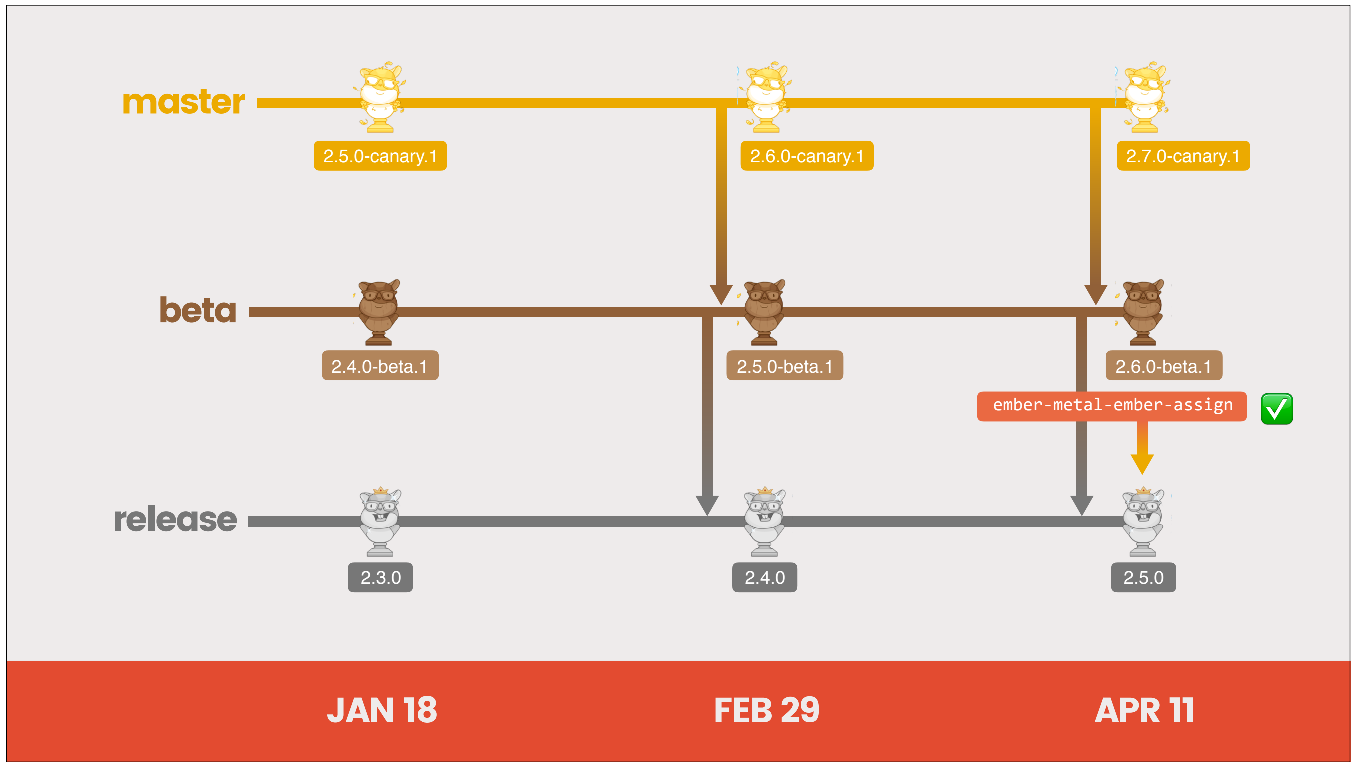


(click) some more time passes, and we're getting ready to branch Ember 2.5 beta from the Ember 2.5 canary branch.

(click) first, we branch the release branch from the previous beta branch.
(click) next, we update the beta branch to master. because the feature is enabled in the features.json, (click) it will be included in the beta build.

(click) now the feature reaches its last test; will it make it through the beta period unscathed? practically speaking, almost every feature that makes it to beta makes it through, but this gives us an opportunity to do one last sanity check with the community. incidentally, the reason features usually make it through is that at this point it's pretty easy to fix any problems that we find during the beta cycle.

(click) some more time passes, and we're ready to release Ember 2.5.
(click) we update the release branch to the latest beta, and since the feature is still enabled, it makes it into the release!



(click) some more time passes, and we're getting ready to branch Ember 2.5 beta from the Ember 2.5 canary branch.

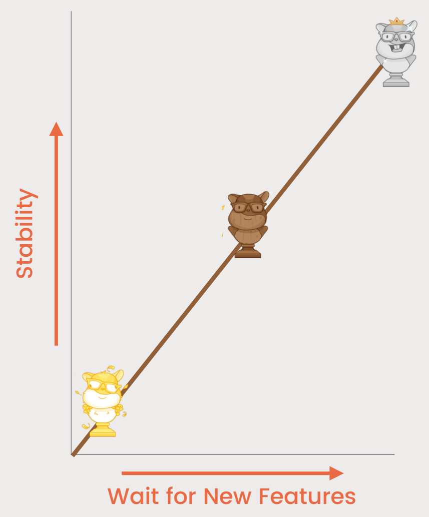
(click) first, we branch the release branch from the previous beta branch.
(click) next, we update the beta branch to master. because the feature is enabled in the features.json, (click) it will be included in the beta build.

(click) now the feature reaches its last test; will it make it through the beta period unscathed? practically speaking, almost every feature that makes it to beta makes it through, but this gives us an opportunity to do one last sanity check with the community. incidentally, the reason features usually make it through is that at this point it's pretty easy to fix any problems that we find during the beta cycle.

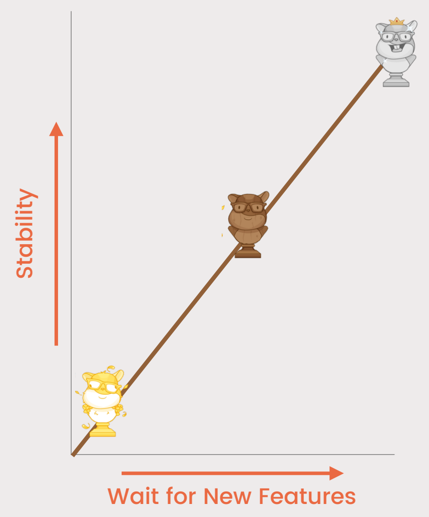
(click) some more time passes, and we're ready to release Ember 2.5.
(click) we update the release branch to the latest beta, and since the feature is still enabled, it makes it into the release!

Now you know the story of the three release channels, and how our process results in a fairly stable framework.

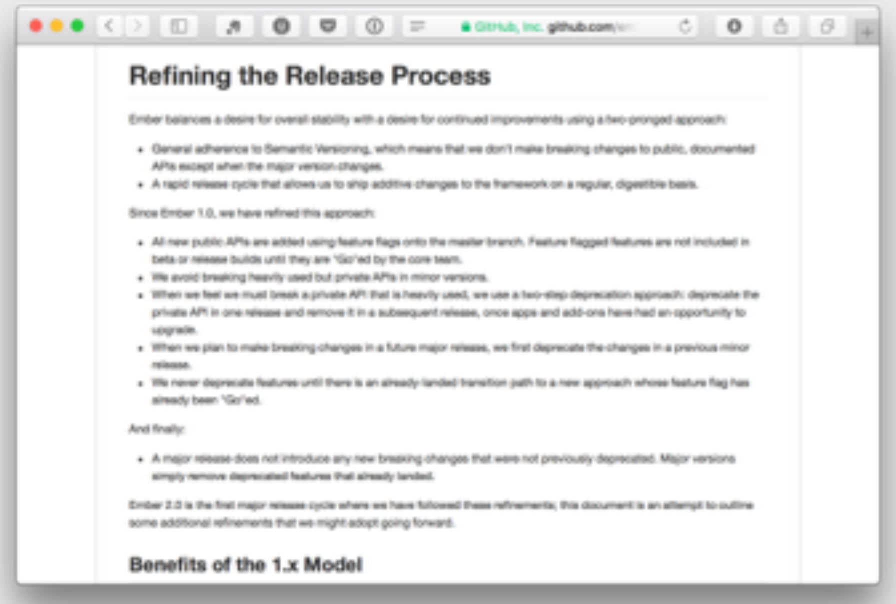
However, over the past year, we learned that while this practically results in a fairly stable and predictable set of changes to the framework, not every application can update every six weeks. That means that applications end up on a whole spectrum of versions, and add-on authors have trouble fully supporting these users.


Stability
Wait for New Features

Upgrade Cadence is Another Axis



RFC 56



GitHub, Inc. github.com/

Refining the Release Process

Ember balances a desire for overall stability with a desire for continued improvements using a two-pronged approach:

- General adherence to Semantic Versioning, which means that we don't make breaking changes to public, documented APIs except when the major version changes.
- A rapid release cycle that allows us to ship additive changes to the framework on a regular, digestible basis.

Since Ember 1.0, we have refined this approach:

- All new public APIs are added using feature flags onto the master branch. Feature flagged features are not included in beta or release builds until they are "Go"ed by the core team.
- We avoid breaking heavily used but private APIs in minor versions.
- When we feel we must break a private API that is heavily used, we use a two-step deprecation approach: deprecate the private API in one release and remove it in a subsequent release, once apps and add-ons have had an opportunity to upgrade.
- When we plan to make breaking changes in a future major release, we first deprecate the changes in a previous minor release.
- We never deprecate features until there is an already-landed transition path to a new approach whose feature flag has already been "Go"ed.

And finally:

- A major release does not introduce any new breaking changes that were not previously deprecated. Major versions simply remove deprecated features that already landed.

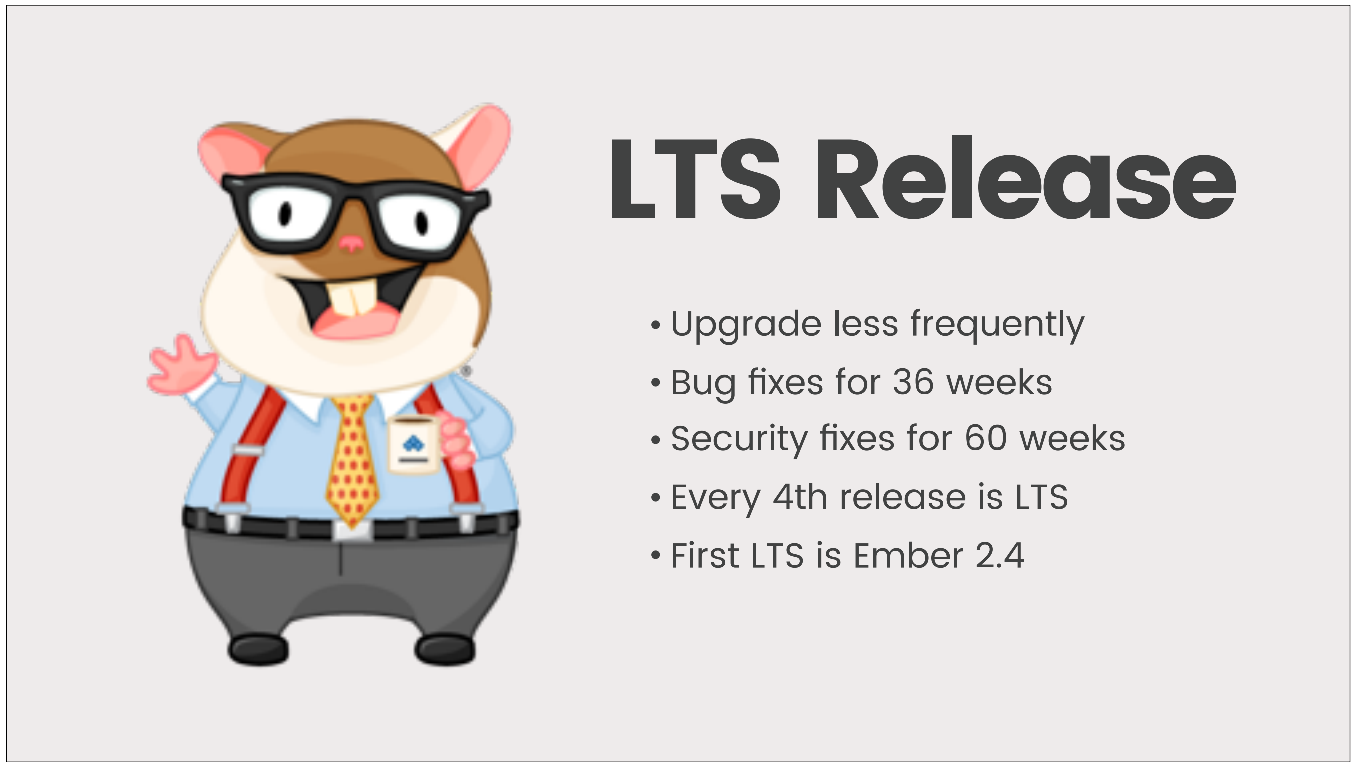
Ember 2.0 is the first major release cycle where we have followed these refinements; this document is an attempt to outline some additional refinements that we might adopt going forward.

Benefits of the 1.x Model

To address these issues, we starting thinking of ways to improve our release process even further. I wrote RFC 56 in May 2015, during the Ember 2.0 canary period.

Among other things, RFC 56 proposed the idea of LTS releases. The important thing about LTS releases is that they're just another channel that gets updated less frequently than the release channel, but still gets official support.

We approved the RFC on October 2, which of course required us to create a new Tomster to represent the LTS channel. (click)

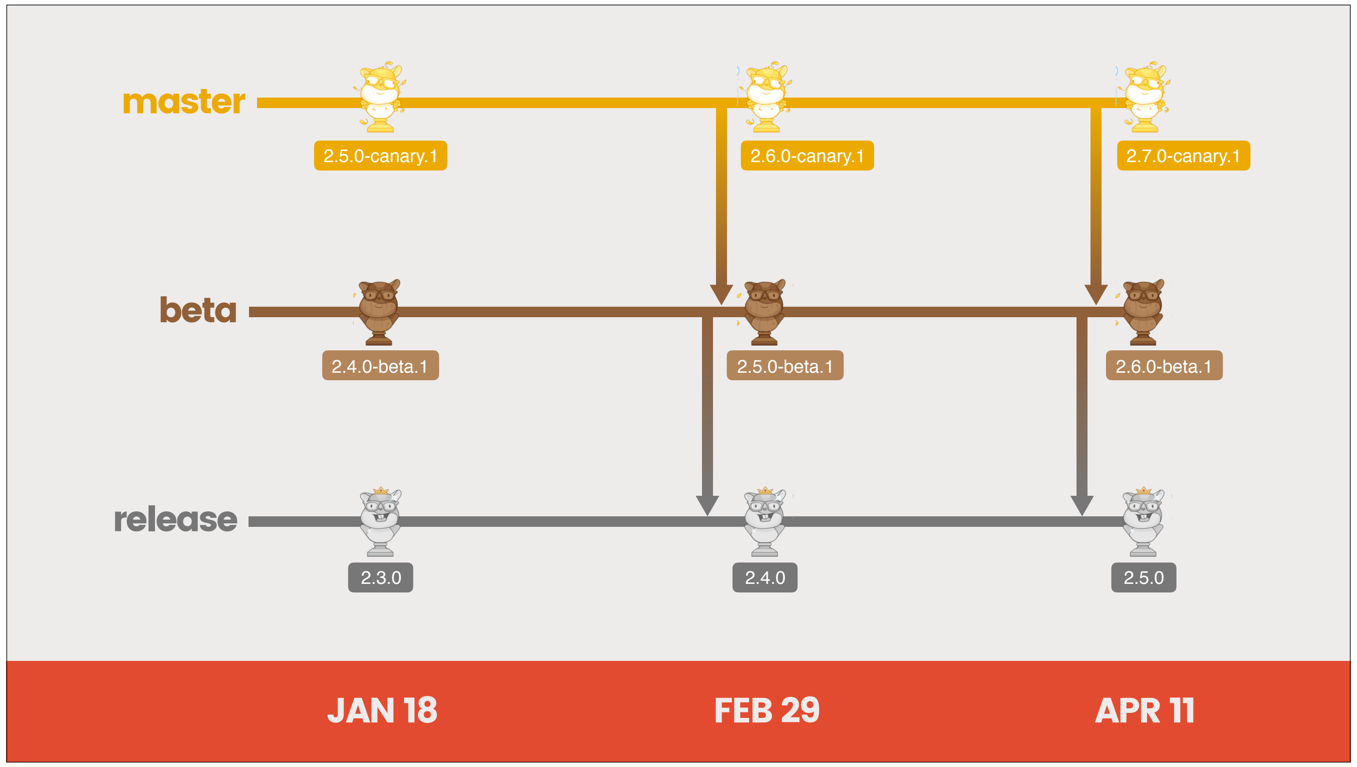


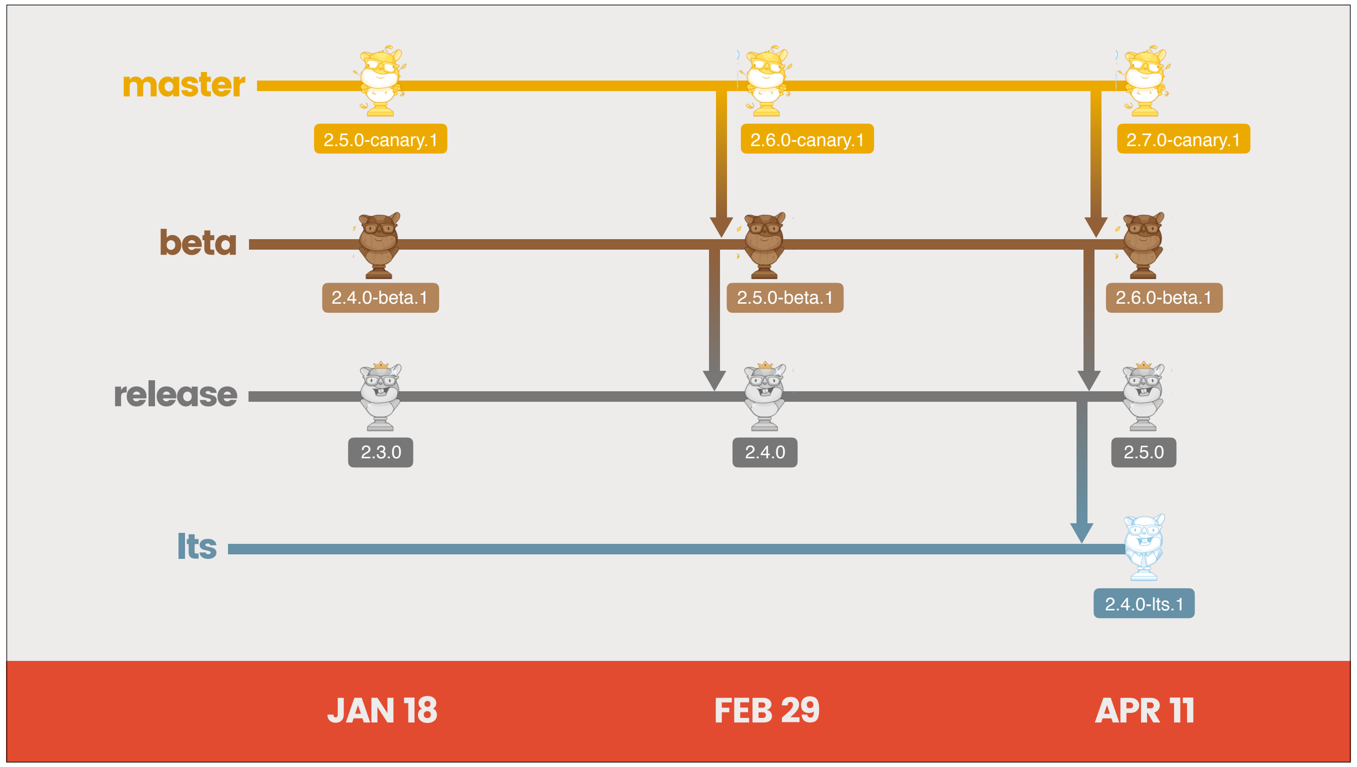
LTS Release

- Upgrade less frequently
- Bug fixes for 36 weeks
- Security fixes for 60 weeks
- Every 4th release is LTS
- First LTS is Ember 2.4

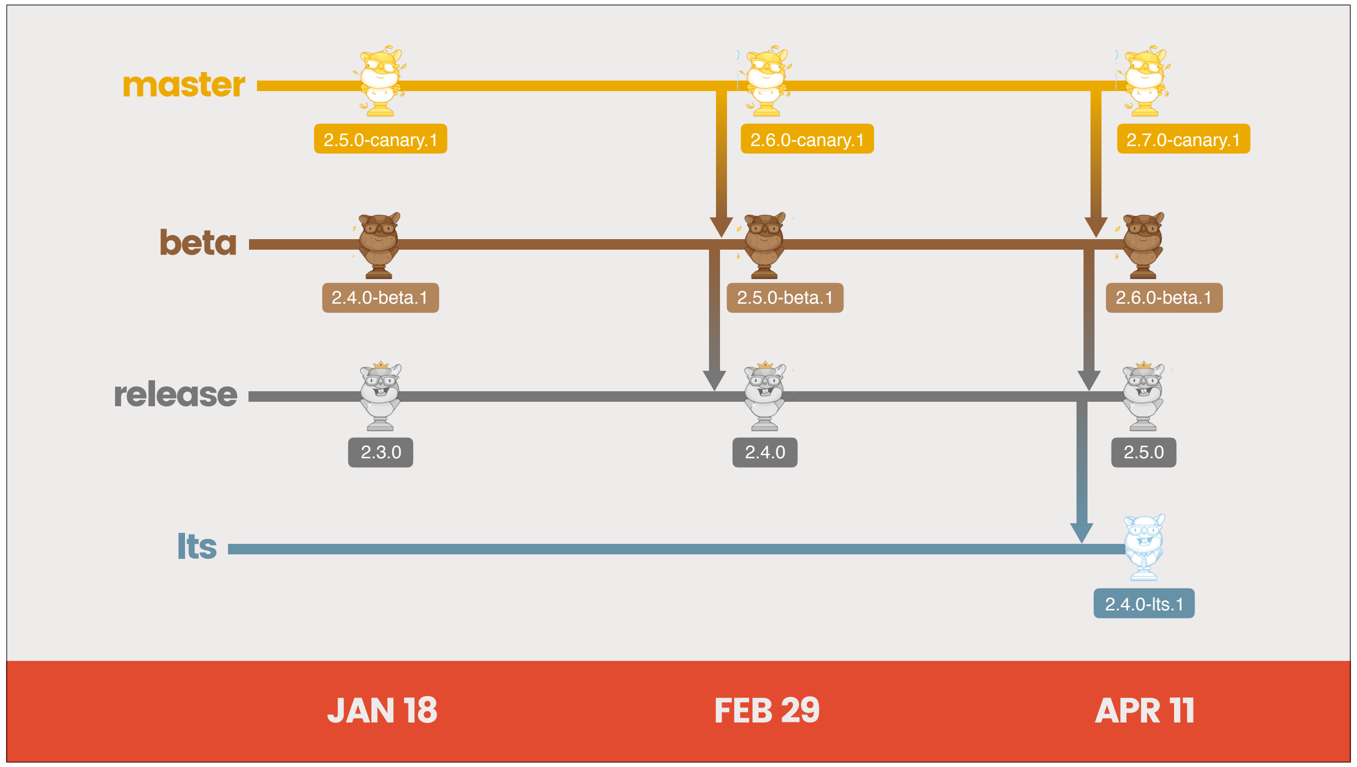
The existing release channels allow you to make a tradeoff. On canary, you get features as quickly as they land, but get no guarantees about those features. On the stable release channel, you have to wait 12 weeks for features to stabilize and make their way through the beta process, but you are rewarded with semver guarantees. While this provides all the flexibility you need to make stability vs. features tradeoff, there is another orthogonal dimension: how often you can schedule time to upgrade. LTS releases give the community an alternative, sanctioned schedule that works better for users who prefer a slower pace.
By synchronizing the timing that these users upgrade, the community can decide to focus energy on specific versions, rather than a scattershot attempt to support every possible combination. This should result in more consistent support and easier upgrades for users on the LTS channel.

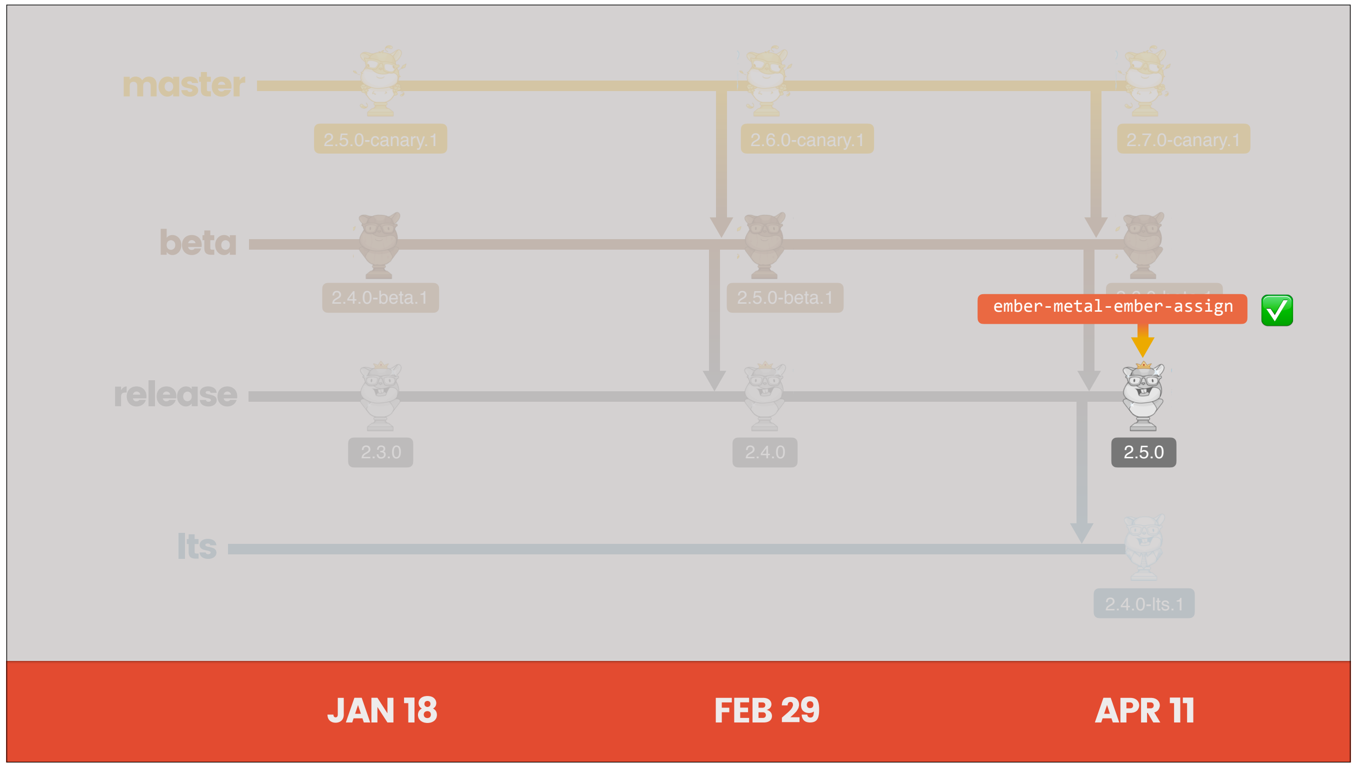
And we now have a fourth release channel bust. To understand how the LTS process works, let's enhance the earlier diagram to include the LTS channel.

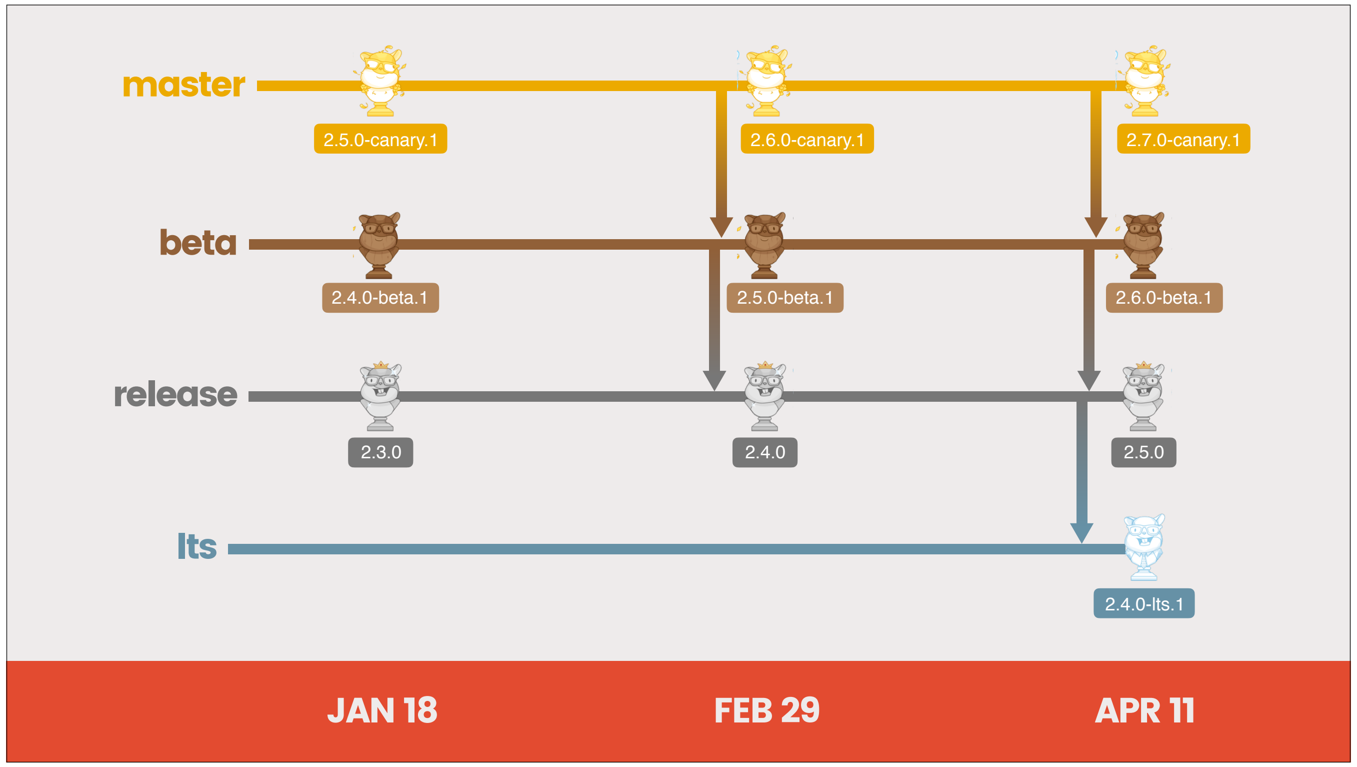

master
2.5.0-canary.1
2.6.0-canary.1
2.7.0-canary.1
beta
2.4.0-beta.1
2.5.0-beta.1
2.6.0-beta.1
release
2.3.0
2.4.0
2.5.0
JAN 18
FEB 29
APR 11



Ok, so now that we have the LTS, what does that mean for ember-metal assign

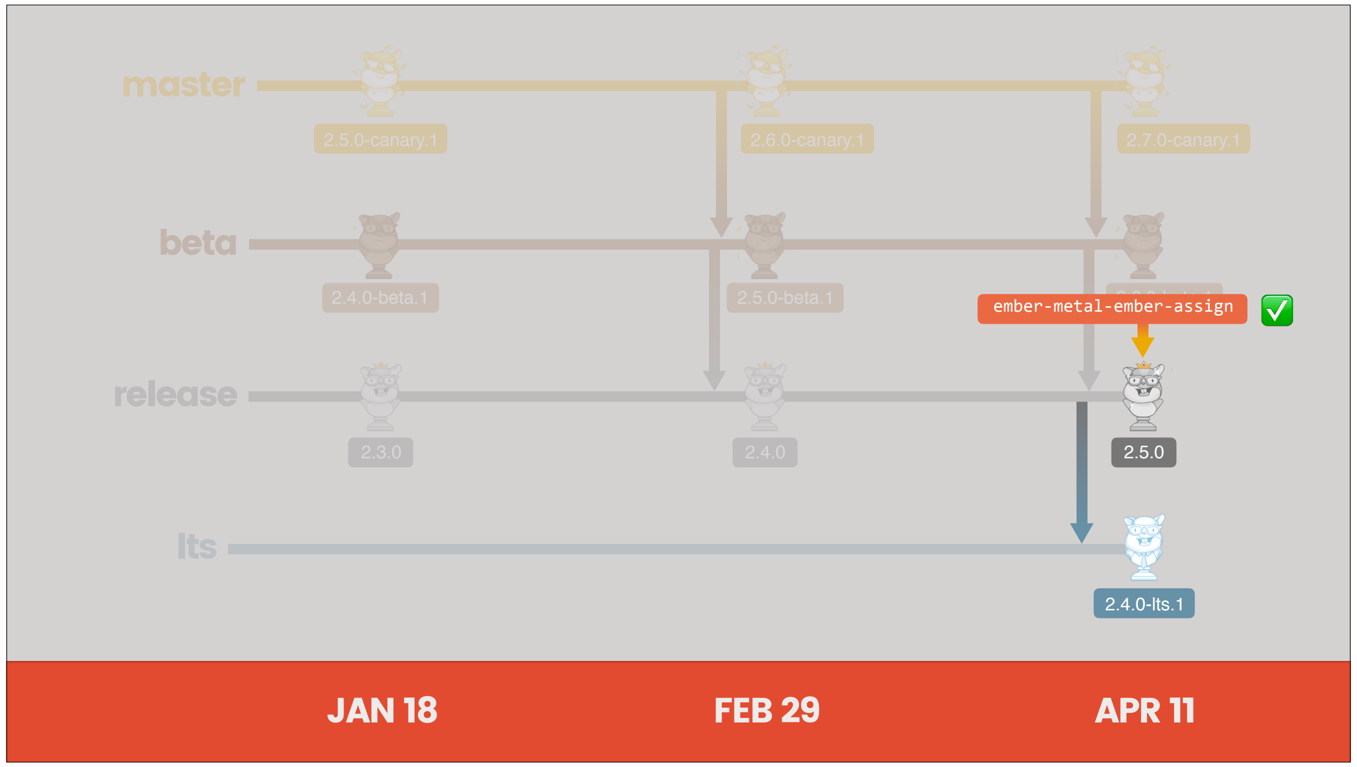

master
2.5.0-canary.1
2.6.0-canary.1
2.7.0-canary.1
beta
2.4.0-beta.1
2.5.0-beta.1
2.6.0-beta.1
release
2.3.0
2.4.0
2.5.0
lts
2.4.0-lts.1
JAN 18
FEB 29
APR 11


master
2.5.0-canary.1
2.6.0-canary.1
2.7.0-canary.1
beta
2.4.0-beta.1
2.5.0-beta.1
ember-metal-ember-assign ✅
release
2.3.0
2.4.0
2.5.0
lts
2.4.0-lts.1
JAN 18
FEB 29
APR 11



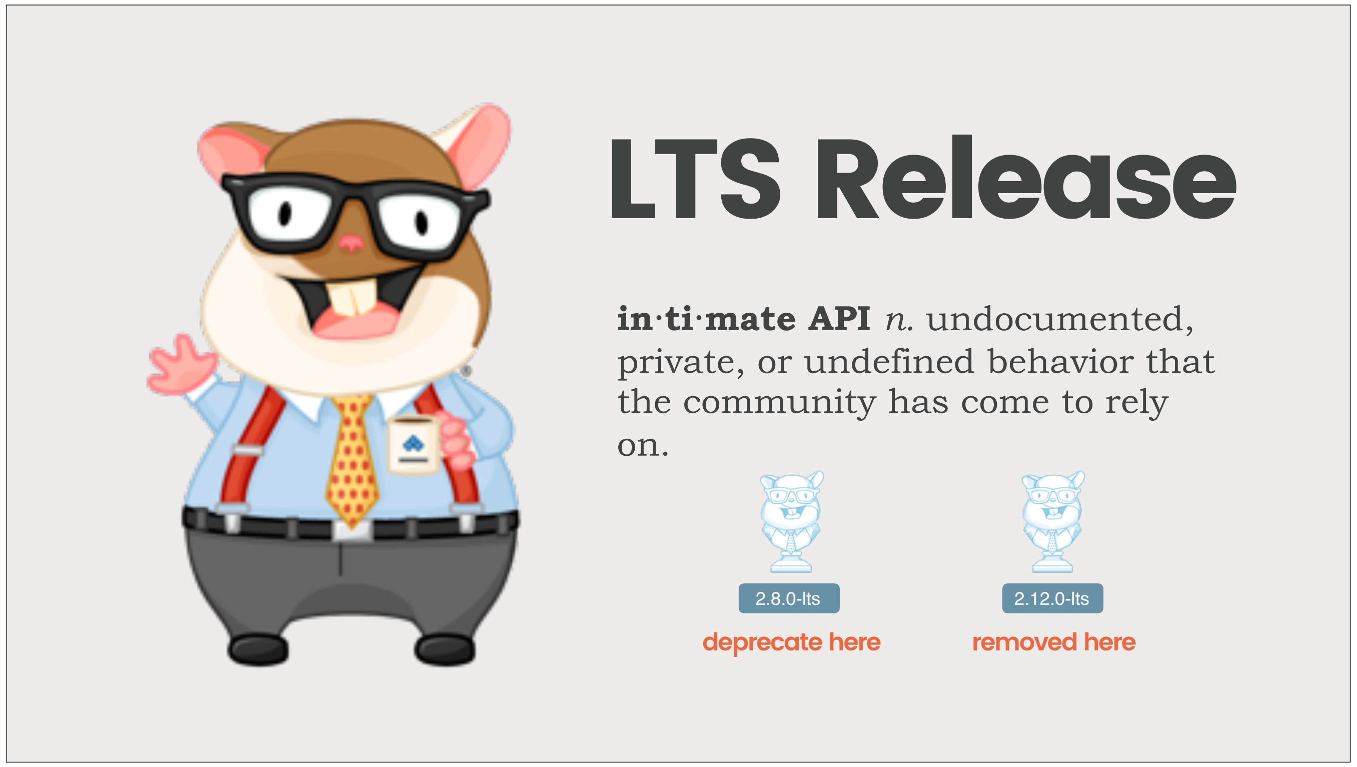
Since it wasn't in 2.4.0, it won't be in 2.4.0-lts. That means that it will land in 2.8.0-lts. The tradeoff in action!

It's worth noting that for features like this that are relatively easy to polyfill, members of the core team usually maintain polyfills that work on older versions of Ember to help add-on authors create addons that work across multiple versions of Ember.



Since it wasn't in 2.4.0, it won't be in 2.4.0-lts. That means that it will land in 2.8.0-lts. The tradeoff in action!

It's worth noting that for features like this that are relatively easy to polyfill, members of the core team usually maintain polyfills that work on older versions of Ember to help add-on authors create addons that work across multiple versions of Ember.



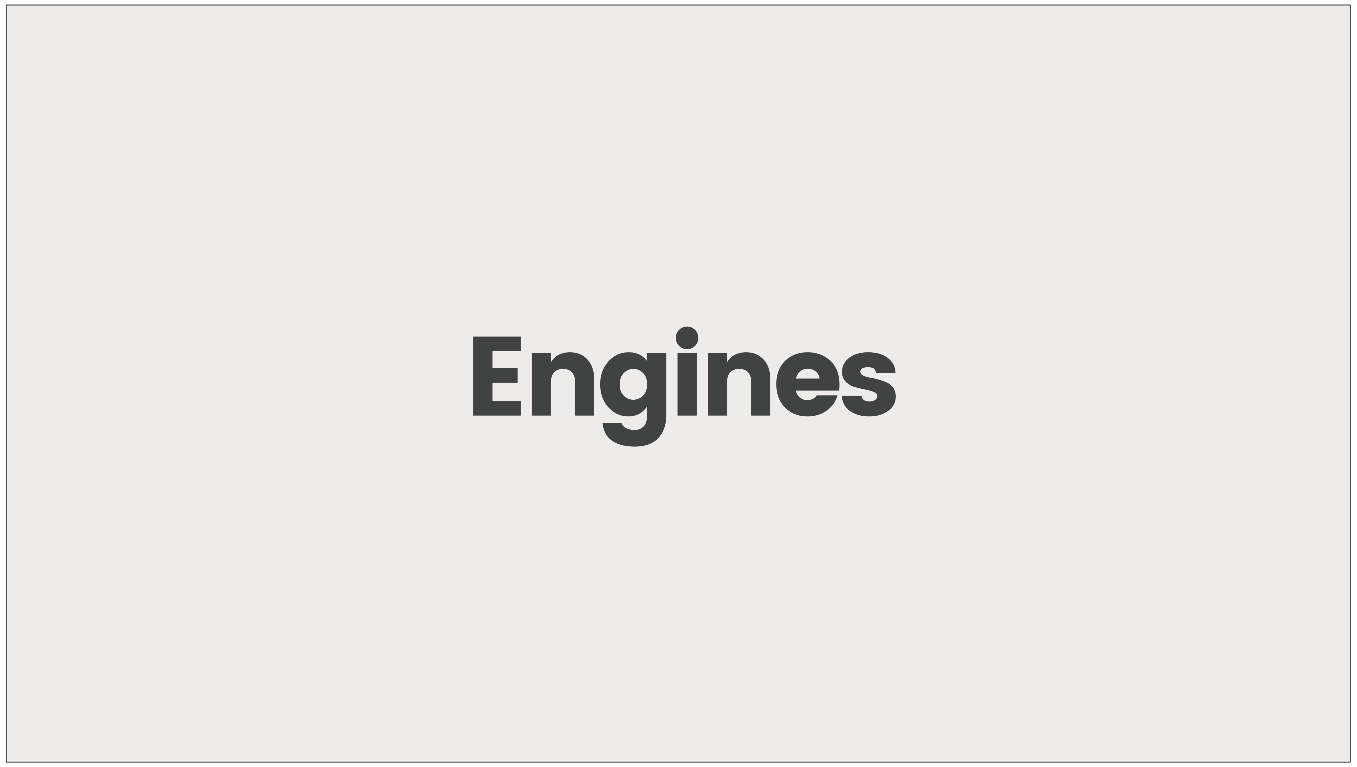
LTS Release

in·ti·mate API *n.* undocumented, private, or undefined behavior that the community has come to rely on.

2.8.0-lts
deprecate here
2.12.0-lts
removed here

Sometimes people come to depend on APIs that weren't public. While these APIs are technically not covered by semver, we still take this breakage seriously. These APIs will not be removed in an LTS release unless they had been deprecated in the previous LTS release. This gives add ons time to transition off of these APIs. If you rely on a lot of intimate APIs, you will probably benefit from the LTS cadence.

Lifecycle of a Big Feature



Engines

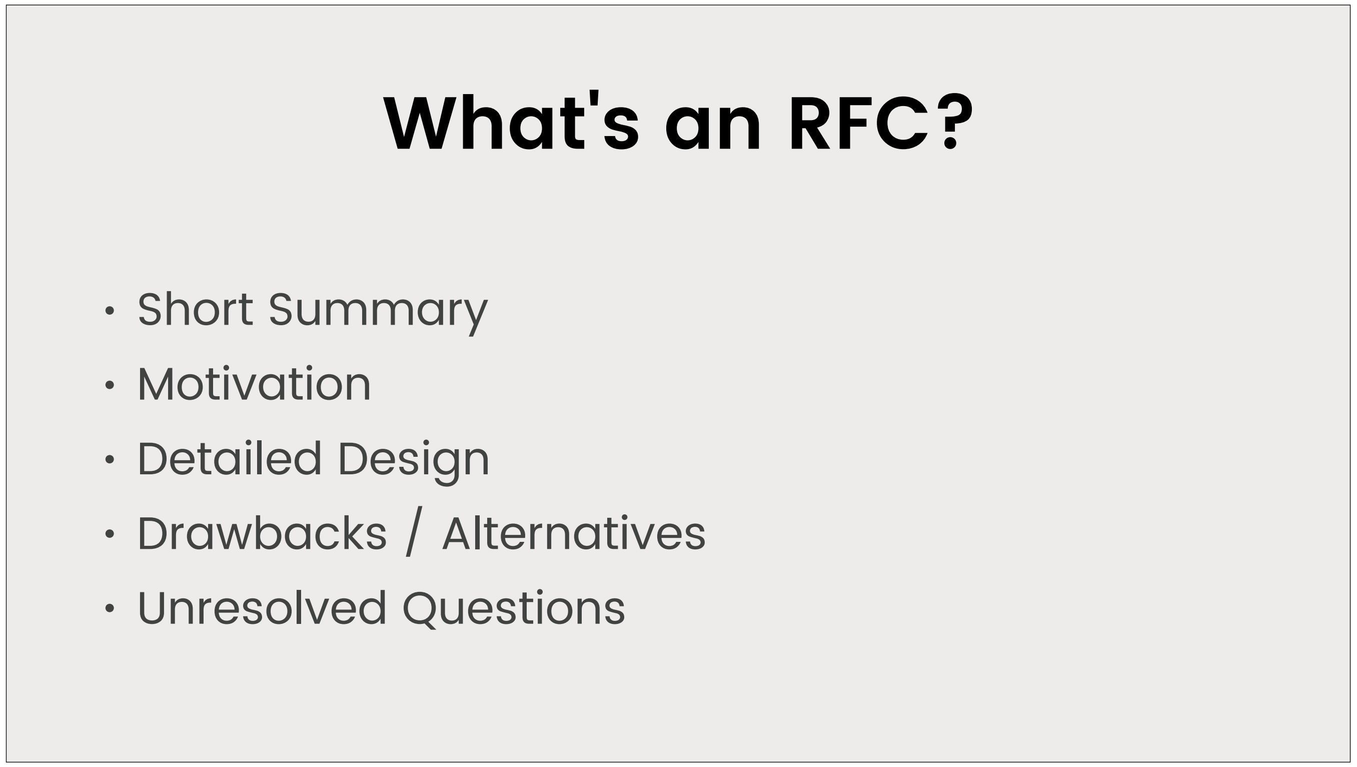
(explain the feature)

RFC: Initial Planning

We talked about RFCs a little bit earlier, but let's dig a little deeper. Any significant feature, even if it's being worked on by a member of the core team or a member of a sub team must submit an RFC. This is not just lip-service; I can't count the number of times I wrote an RFC and changed it significantly due to feedback from one or more interested community members. It's almost important to make sure there is broad consensus about the direction of the framework, which is impossible unless the rationale the core team uses to make decisions is presented and debated publicly. (no new rationale rule)

If you're keeping track, here's what the whole lifecycle looks like.

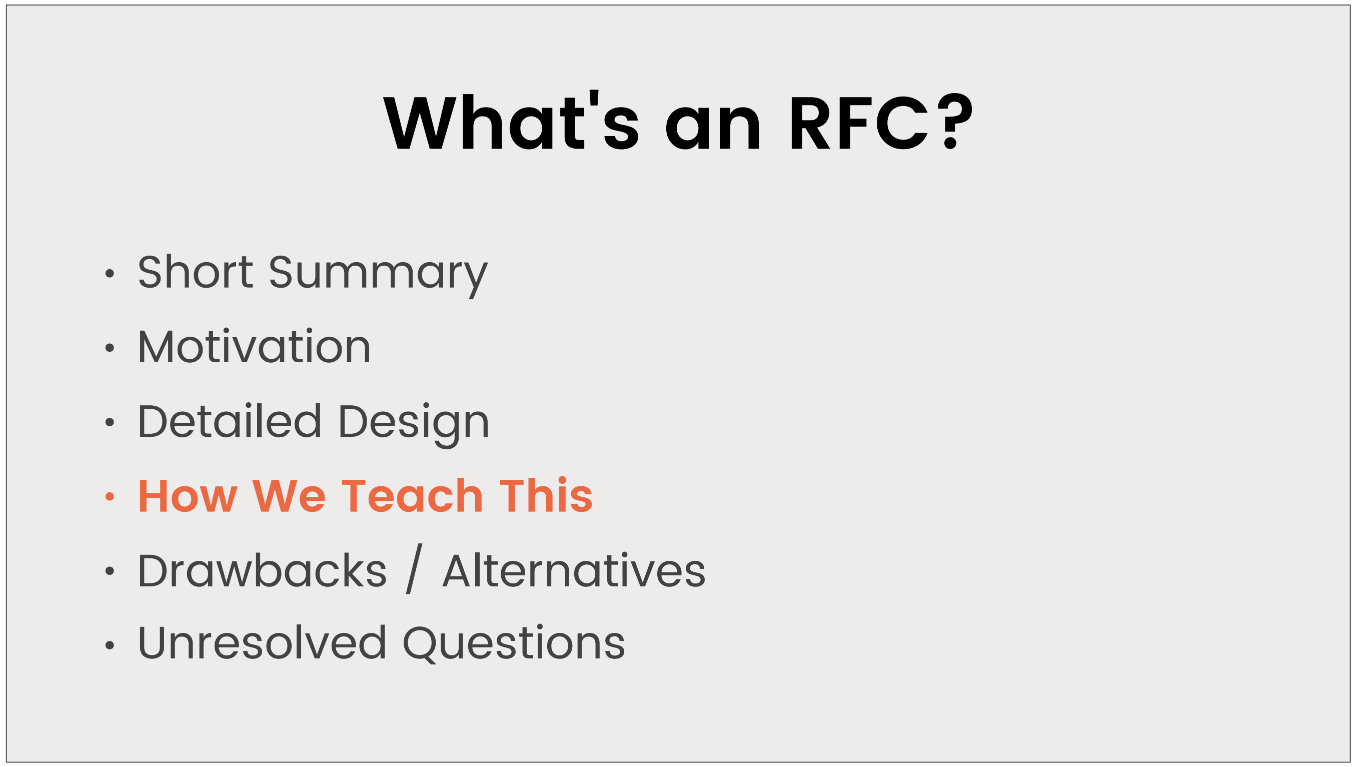
So what's in an RFC?



What's an RFC?

- Short Summary
- Motivation
- Detailed Design
- Drawbacks / Alternatives
- Unresolved Questions

This was the original template, like I said, cribbed directly from Rust. Worth calling out: the process of doing even pro-forma drawbacks and alternatives is a really great process. I always find myself discovering something I hadn't considered when I'm forced to stop and think about alternatives, even if they're non-starters. In fact, the process of arguing against non-starters is sometimes harder than it looks.



What's an RFC?

- Short Summary
- Motivation
- Detailed Design
- **How We Teach This**
- Drawbacks / Alternatives
- Unresolved Questions

Recently, we added a new section to the RFC template: "How we teach this".

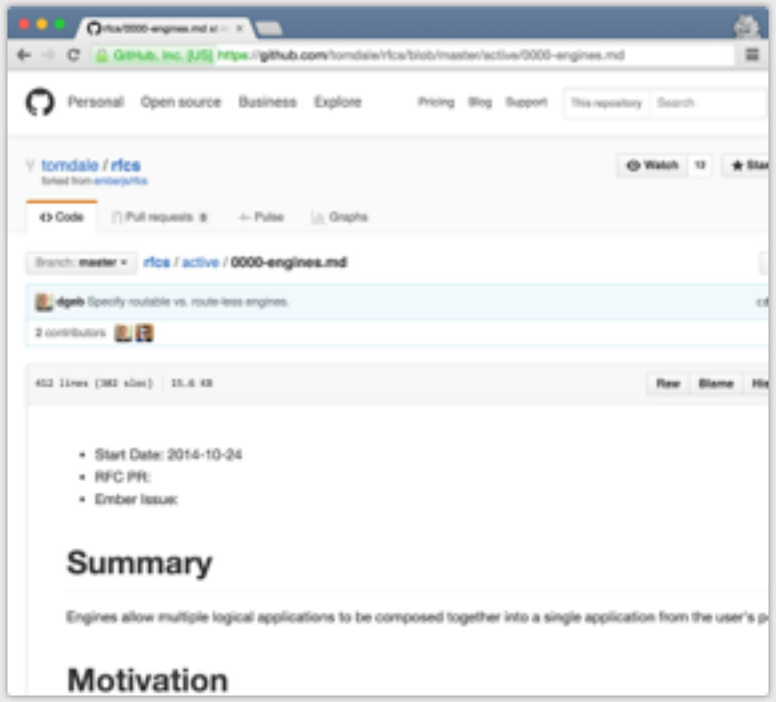
What **names and terminology** work best for these concepts and why? How is this idea best presented? As a continuation of existing Ember patterns, or as a wholly new one?

Would the acceptance of this proposal mean the Ember guides must be re-organized or altered? **Does it change how Ember is taught to new users at any level?**

How should this feature be introduced and **taught to existing Ember users?**

The idea behind this section is threefold:

1. We noticed that the terminology in the RFC got spread pretty far, even if it was just used to help clarify implementation details. This is in large because RFC participants tend to be influencers, and if there is no alternative…
2. We felt that while the RFC process was doing a good job of helping us maintain control of technical complexity, it took a lot of work (not always in the RFC process) to maintain control of programming model complexity. Making it an explicit part of the process helps.
3. There are also many users with a good understanding of the programming model who aren't very interested in the technical nitty gritty. This gives them a chance to contribute, and us a chance to learn from them.



https://github.com/tomdale/rfcs/blob/master/active/0000-engines.md

Personal Open source Business Explore Pricing Blog Support

tomdale / rfcs
forked from emberjs/rfcs

Code Pull requests Pulse Graphs

Branch: master rfcs / active / 0000-engines.md

dgeb Specify routable vs. route-less engines.

2 contributors

412 lines (382 sloc) 15.6 KB

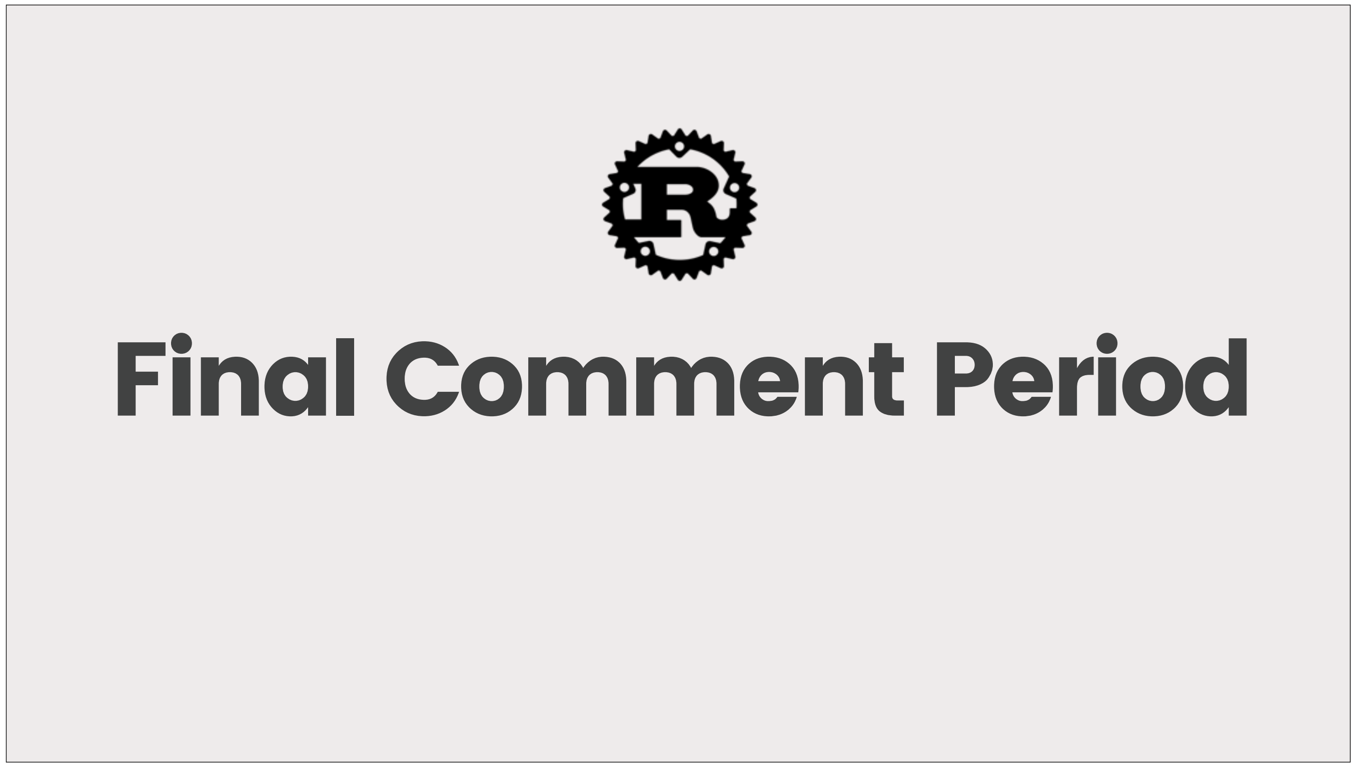
- Start Date: 2014-10-24
- RFC PR:
- Ember Issue:

Summary

Engines allow multiple logical applications to be composed together into a single application from the user's p

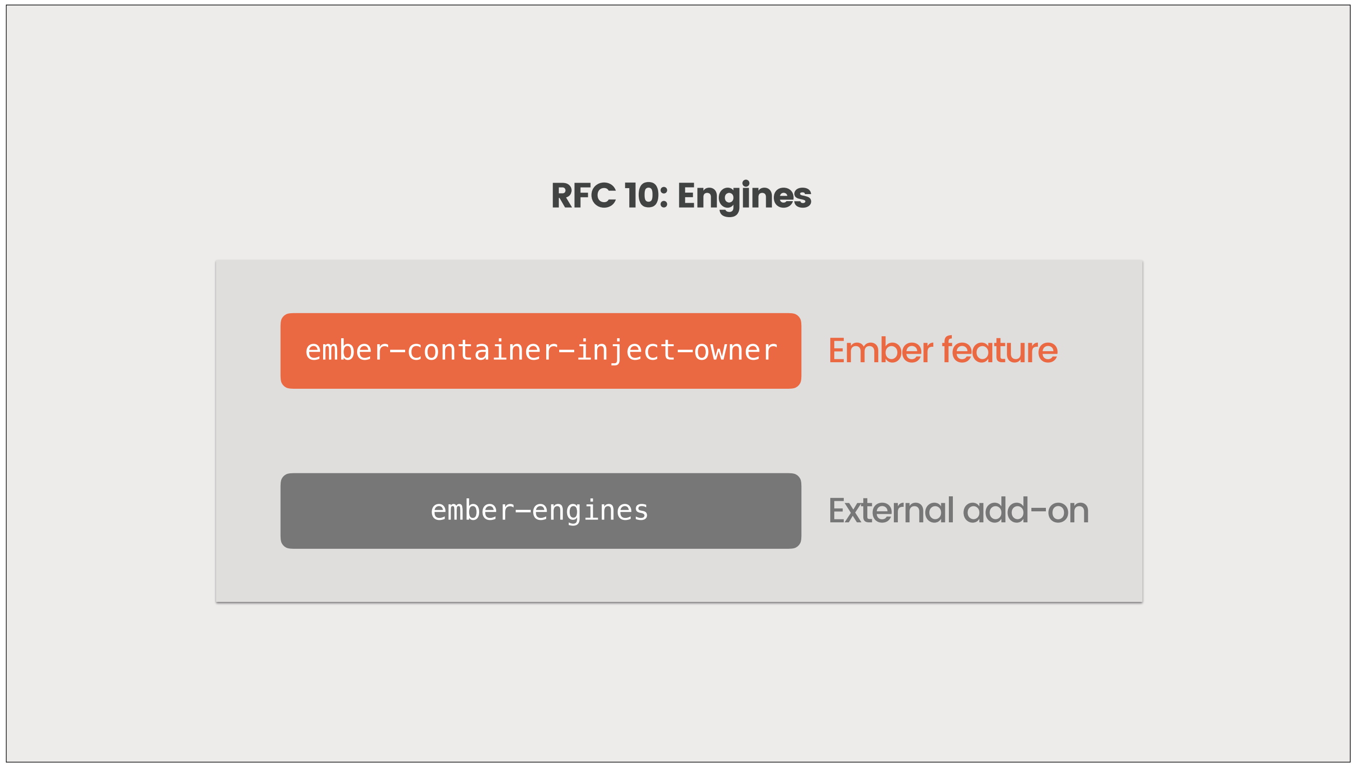
Motivation

In this case, the original RFC for engines was written by Tom (with some help from me) in October 2014. It has, to date, 199 comments, many of which significantly altered the trajectory of the feature.



Final Comment Period

Rust does a week-long FCP before merging an RFC. We should do this in Ember.



RFC 10: Engines

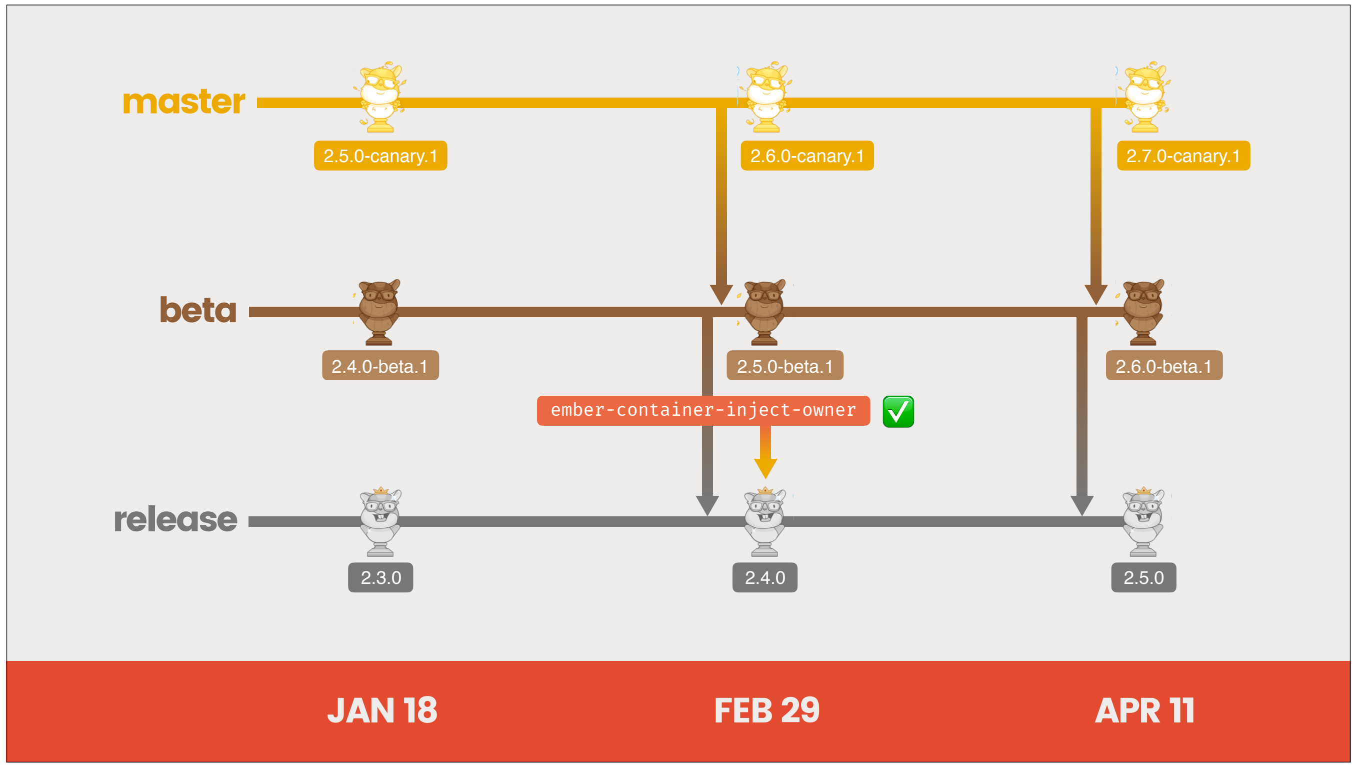
`ember-container-inject-owner` Ember feature

`ember-engines` External add-on

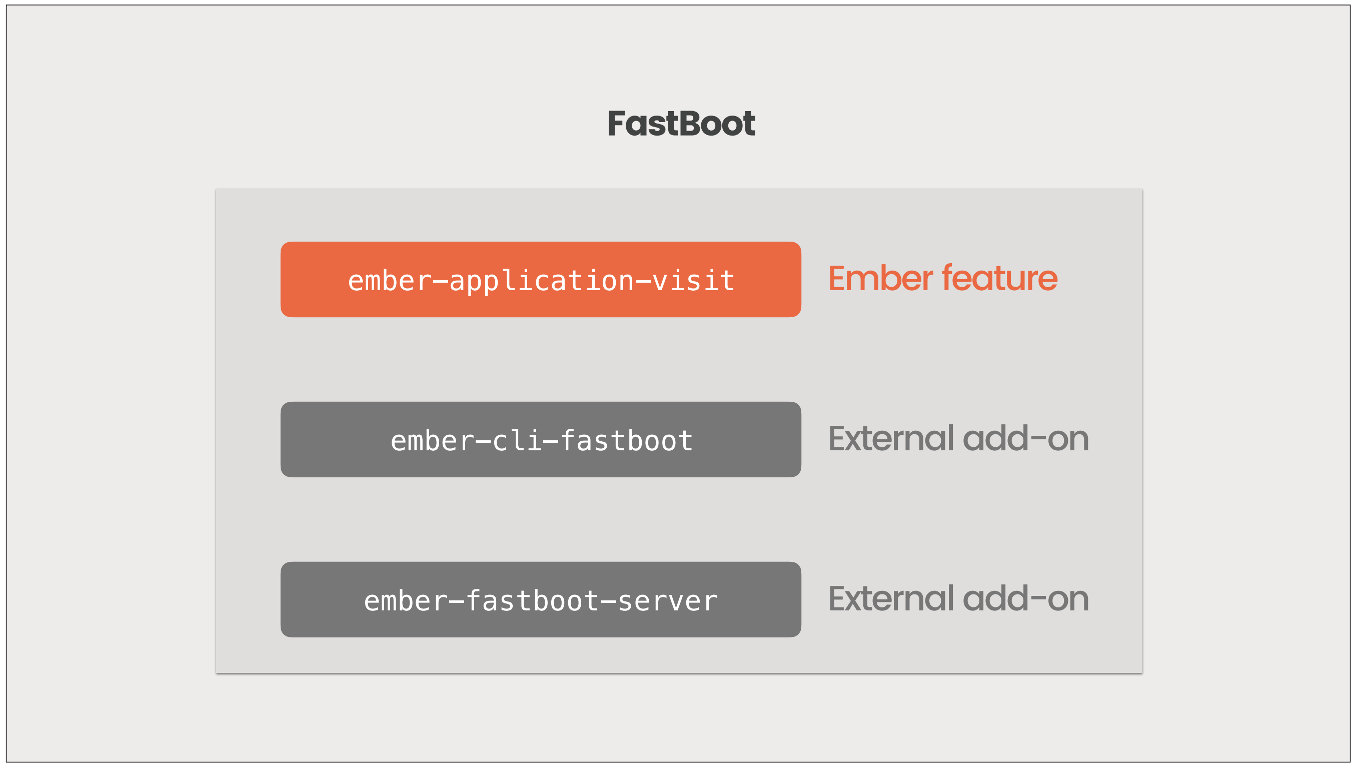
Once a RFC is accepted, it is usually broken up into parts. In the case of engines, it was broken up into pieces. In the case of engines, Dan implemented the critical capabilities needed to enable engines into the ember-container-inject-owner as an Ember feature, and implemented the more experimental parts of the API in an external add-on. While the feature could in principle be developed behind a feature flag, this allows us to experiment with bigger-picture engine questions in a package with its own stability story, built on top of a stable core that we can ship quickly as part of Ember's compatibility story.

Among other things, this means that you can keep using an older version of the engines add-on as it progresses and as it makes its way into Ember.

After the RFC is merged, the process is identical to `ember-assign`

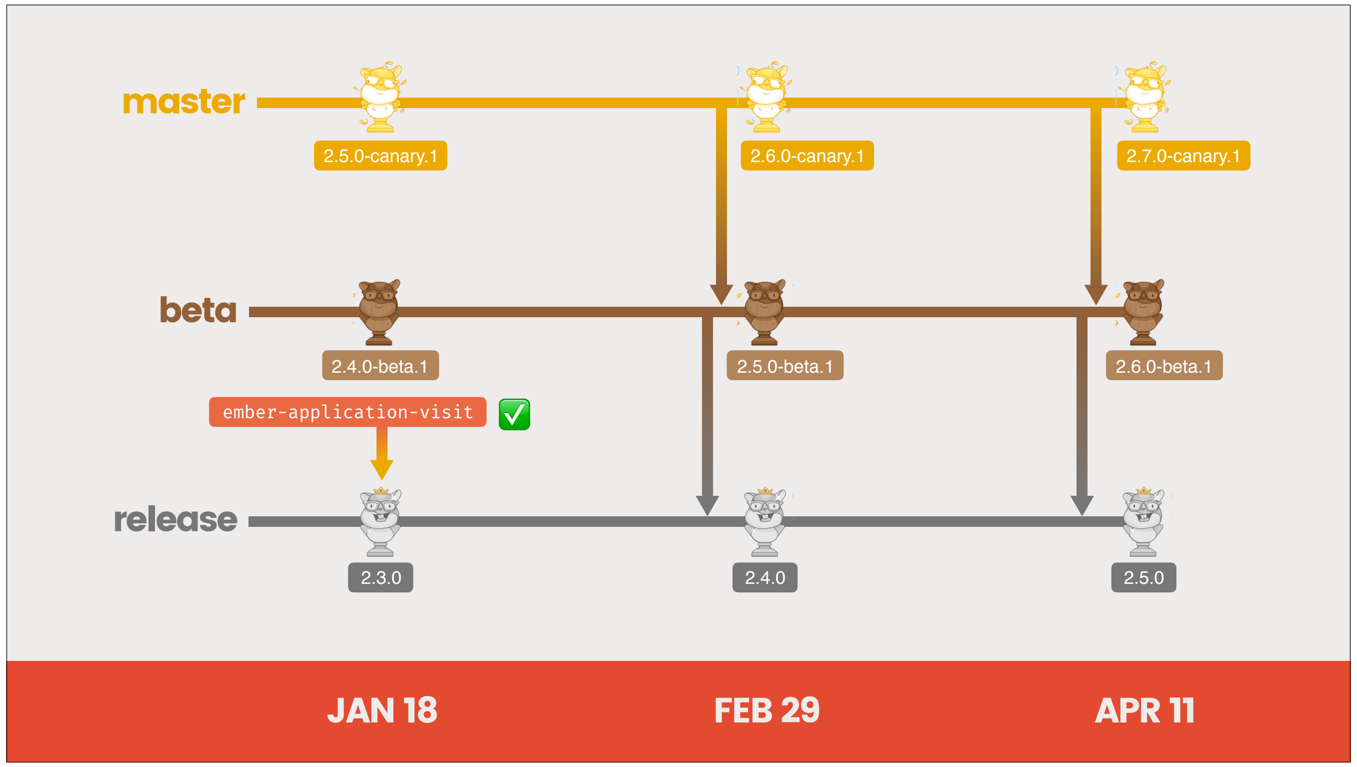


ember-container-inject-owner landed in Ember 2.4, and people are already starting to put the engines add-on into production :)

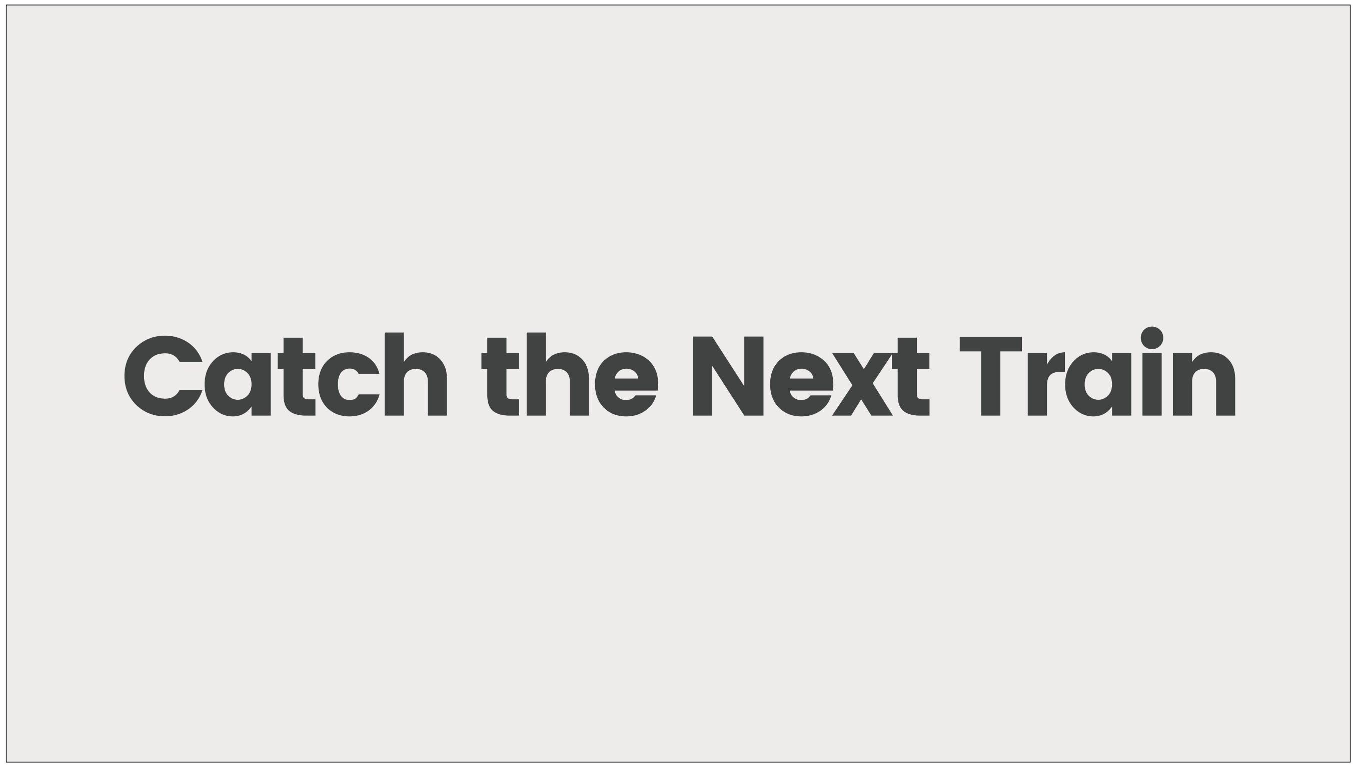


Incidentally The FastBoot feature works the same way. We made the necessary core changes to Ember but are developing the experimental feature outside of Ember core.

(talk about why this is a good idea)



and ember-application-visit landed in 2.3, and people have been using FastBoot in production for a while.



Catch the Next Train

This process is incredibly freeing.

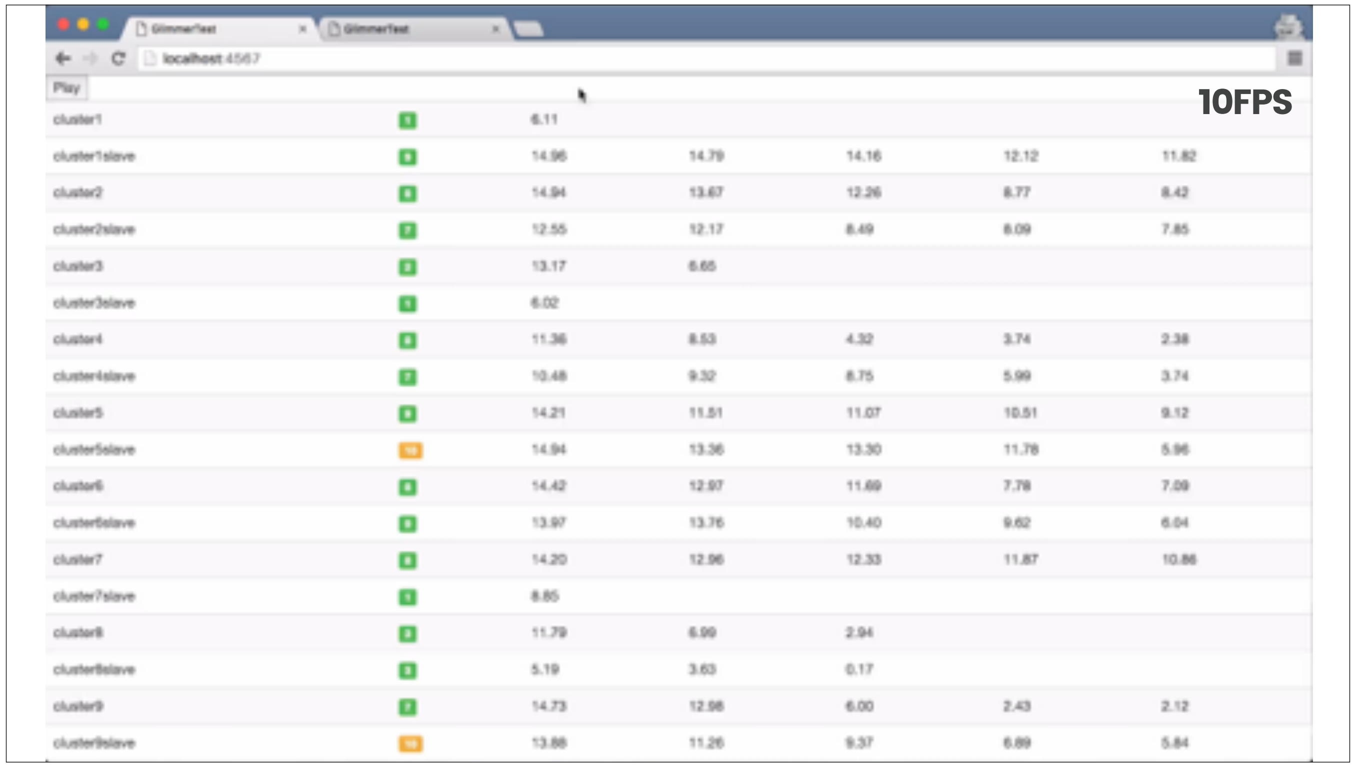
There are benefits to:

* Maintainers: there's no rush to get features in for a given release. If we miss a release, there's another one in six weeks. This means it's relatively easy (as these things go) to push back on shipping not-ready features (which I've never experienced before; it's really freeing)
* Contributors have a predictable way to understand when their feature will make it in and what the process is. They also don't have to sacrifice weekends to make it for a particular release, since the next one is coming.
* Add-on authors and the wider ecosystem can build robust abstractions on top of APIs they know are stable and won't change, and have a clear way of communicating with their users about where their add-on will work (measured in years not months). Add-on authors can also use experimental features to build new functionality and communicate clearly about the stability level of the underlying kernel feature simply based on whether the user is using canary or not.
* And of course, end developers can make a decision about their risk tolerance easily, and are freed up to innovate in their application, which is the best part of this whole process.

Glimmer 2

So all of that explains how you can do small and medium-sized features, but the critique of incrementalism is that you can't ever do really big re-thinks. You might have heard of Glimmer 2. Before I talk about how we manage the process, let's look at what the feature is.

(Unfortunately I can't embed videos in PDFs :( )



GlimmerTest | GlimmerTest

localhost:4567

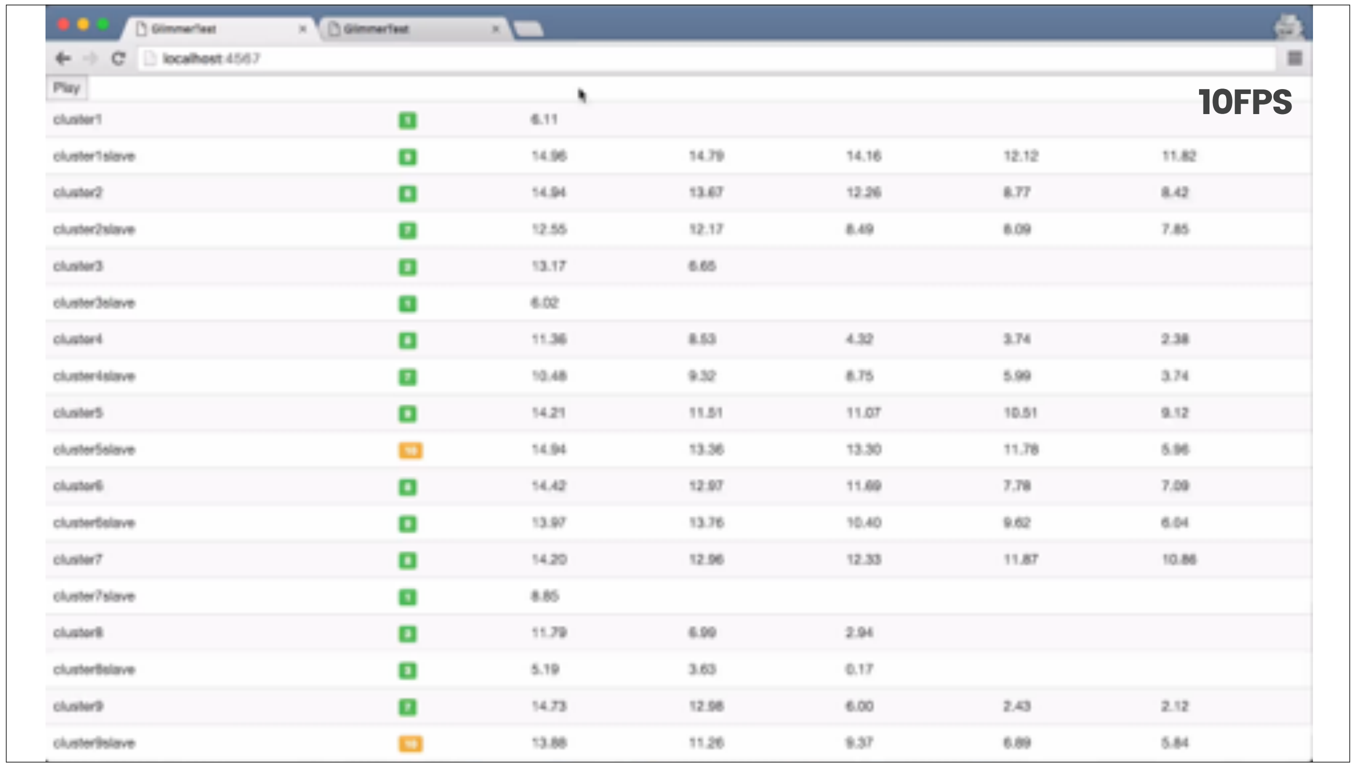
Play

10FPS

cluster1	1	6.11				
cluster1slave	[illegible]	14.96	14.79	14.16	12.12	11.82
cluster2	[illegible]	14.94	13.67	12.26	8.77	8.42
cluster2slave	7	12.55	12.17	8.49	8.09	7.85
cluster3	2	13.17	6.65			
cluster3slave	1	6.02				
cluster4	[illegible]	11.36	8.53	4.32	3.74	2.38
cluster4slave	7	10.48	9.32	8.75	5.99	3.74
cluster5	[illegible]	14.21	11.51	11.07	10.51	9.12
cluster5slave	10	14.94	13.36	13.30	11.78	5.96
cluster6	[illegible]	14.42	12.97	11.69	7.78	7.09
cluster6slave	[illegible]	13.97	13.76	10.40	9.62	6.04
cluster7	[illegible]	14.20	12.96	12.33	11.87	10.86
cluster7slave	1	8.85				
cluster8	3	11.79	6.99	2.94		
cluster8slave	3	5.19	3.63	0.17		
cluster9	7	14.73	12.98	6.00	2.43	2.12
cluster9slave	10	13.88	11.26	9.37	6.89	5.84

You might remember the famous dbmon benchmark from last year…
100 rows, pretty intense…

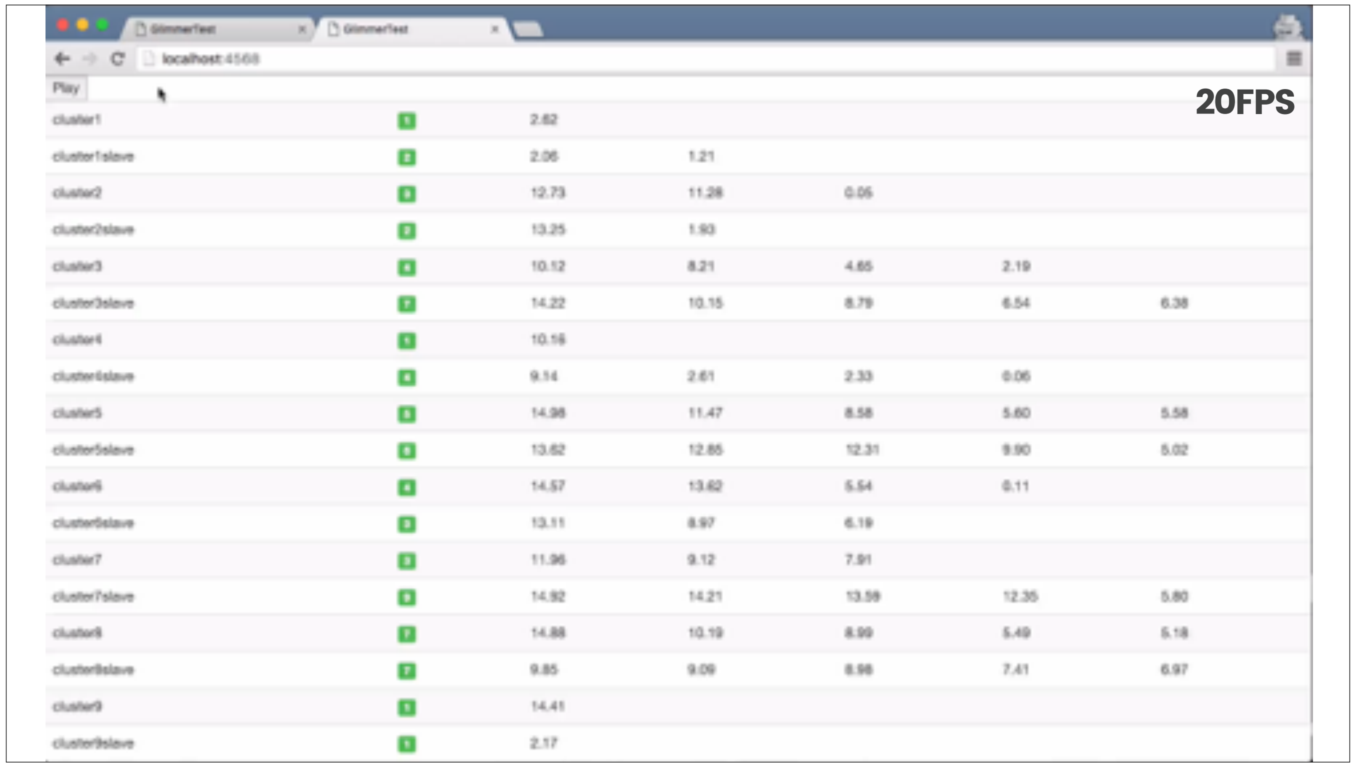
This is DB mon running against Ember 2.6 Canary with HTMLBars (“Glimmer 1”)
It's pretty fast (we’ve racked up some wins since 1.13 last year)



cluster1	1	6.11				
cluster1slave	[illegible]	14.96	14.79	14.16	12.12	11.82
cluster2	[illegible]	14.94	13.67	12.26	8.77	8.42
cluster2slave	[illegible]	12.55	12.17	8.49	8.09	7.85
cluster3	[illegible]	13.17	6.65			
cluster3slave	1	6.02				
cluster4	[illegible]	11.36	8.53	4.32	3.74	2.38
cluster4slave	[illegible]	10.48	9.32	8.75	5.99	3.74
cluster5	[illegible]	14.21	11.51	11.07	10.51	9.12
cluster5slave	10	14.94	13.36	13.30	11.78	5.96
cluster6	[illegible]	14.42	12.97	11.69	7.78	7.09
cluster6slave	[illegible]	13.97	13.76	10.40	9.62	6.04
cluster7	[illegible]	14.20	12.96	12.33	11.87	10.86
cluster7slave	1	8.85				
cluster8	[illegible]	11.79	6.99	2.94		
cluster8slave	[illegible]	5.19	3.63	0.17		
cluster9	[illegible]	14.73	12.98	6.00	2.43	2.12
cluster9slave	10	13.88	11.26	9.37	6.89	5.84

You might remember the famous dbmon benchmark from last year…
100 rows, pretty intense…

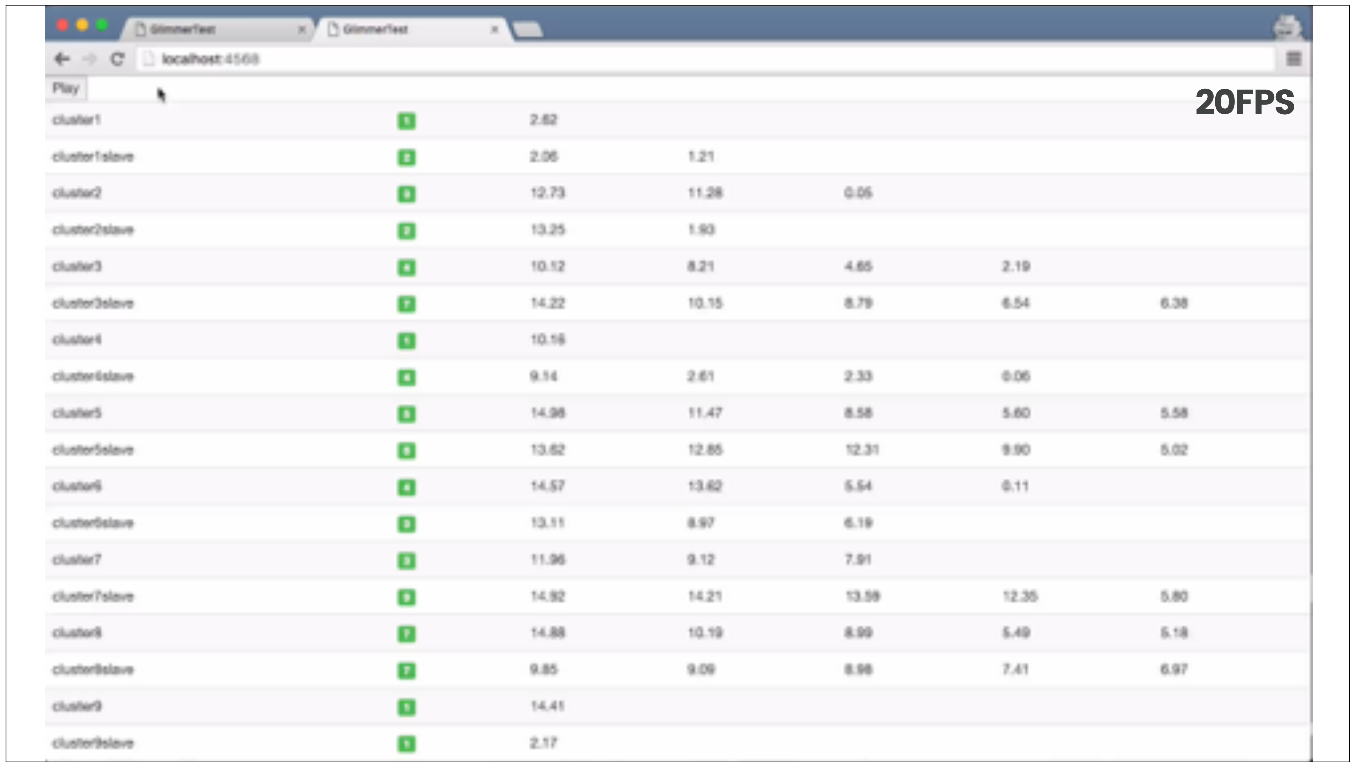
This is DB mon running against Ember 2.6 Canary with HTMLBars ("Glimmer 1")
It's pretty fast (we've racked up some wins since 1.13 last year)



cluster1	1	2.62				
cluster1slave	2	2.06	1.21			
cluster2	3	12.73	11.28	0.05		
cluster2slave	2	13.25	1.93			
cluster3	4	10.12	8.21	4.65	2.19	
cluster3slave	7	14.22	10.15	8.79	6.54	6.38
cluster4	1	10.16				
cluster4slave	4	9.14	2.61	2.33	0.06	
cluster5	5	14.98	11.47	8.58	5.60	5.58
cluster5slave	6	13.62	12.85	12.31	9.90	5.02
cluster6	4	14.57	13.62	5.54	0.11	
cluster6slave	3	13.11	8.97	6.19		
cluster7	3	11.96	9.12	7.91		
cluster7slave	9	14.92	14.21	13.59	12.35	5.80
cluster8	7	14.88	10.19	8.99	5.49	5.18
cluster8slave	7	9.85	9.09	8.98	7.41	6.97
cluster9	1	14.41				
cluster9slave	1	2.17				

This is the same demo running against Ember 2.6 Canary with Glimmer 2.
As you can see, it's twice as fast without any code changes.

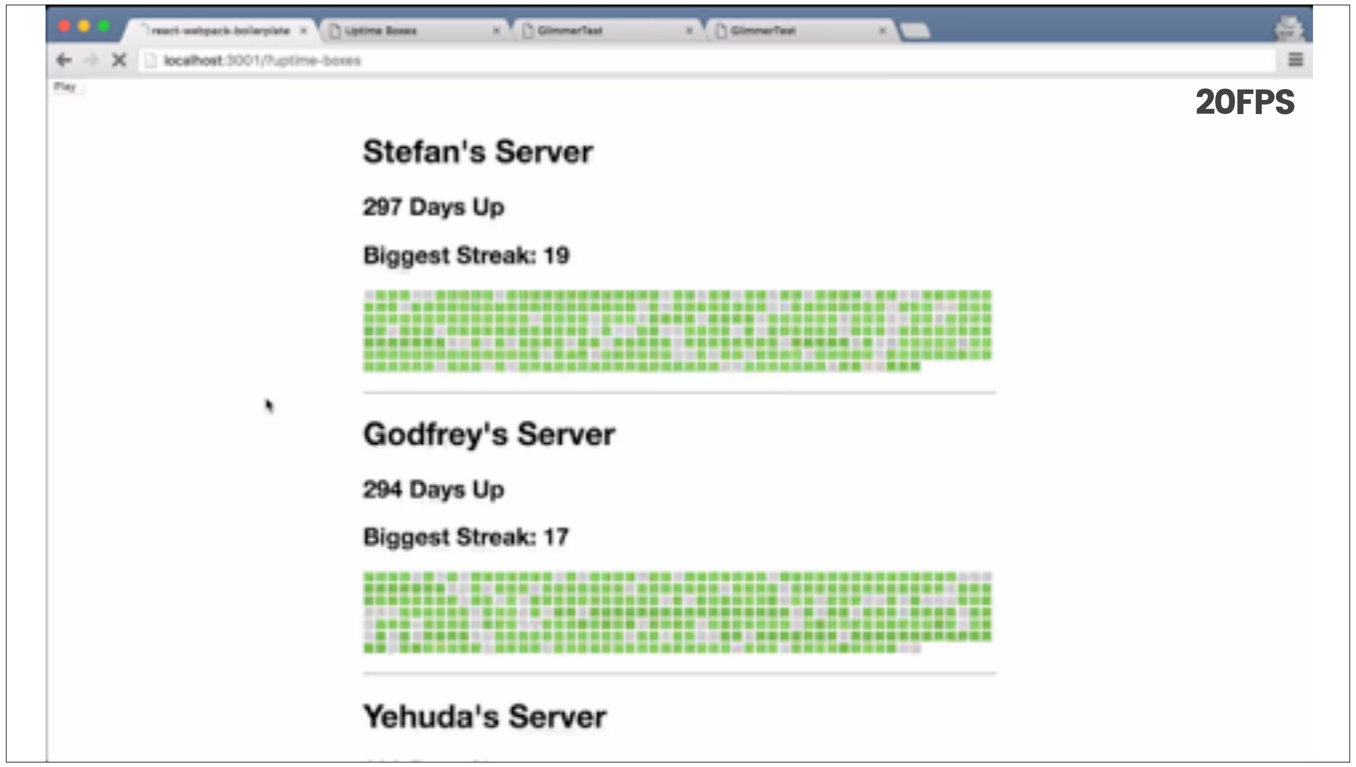
But while dbmon makes a pretty good stress test, it doesn’t capture enough of the way Ember users build apps. In particular, there aren’t enough components in it. **So we made a new benchmark.**



cluster1	1	2.62				
cluster1slave	2	2.06	1.21			
cluster2	3	12.73	11.28	0.05		
cluster2slave	2	13.25	1.93			
cluster3	4	10.12	8.21	4.65	2.19	
cluster3slave	7	14.22	10.15	8.79	6.54	6.38
cluster4	1	10.16				
cluster4slave	4	9.14	2.61	2.33	0.06	
cluster5	5	14.98	11.47	8.58	5.60	5.58
cluster5slave	8	13.62	12.85	12.31	9.90	5.02
cluster6	4	14.57	13.62	5.54	0.11	
cluster6slave	3	13.11	8.97	6.19		
cluster7	3	11.96	9.12	7.91		
cluster7slave	9	14.92	14.21	13.59	12.35	5.80
cluster8	7	14.88	10.19	8.99	5.49	5.18
cluster8slave	7	9.85	9.09	8.98	7.41	6.97
cluster9	1	14.41				
cluster9slave	1	2.17				

This is the same demo running against Ember 2.6 Canary with Glimmer 2.
As you can see, it's twice as fast without any code changes.

But while dbmon makes a pretty good stress test, it doesn’t capture enough of the way Ember users build apps. In particular, there aren’t enough components in it. **So we made a new benchmark.**

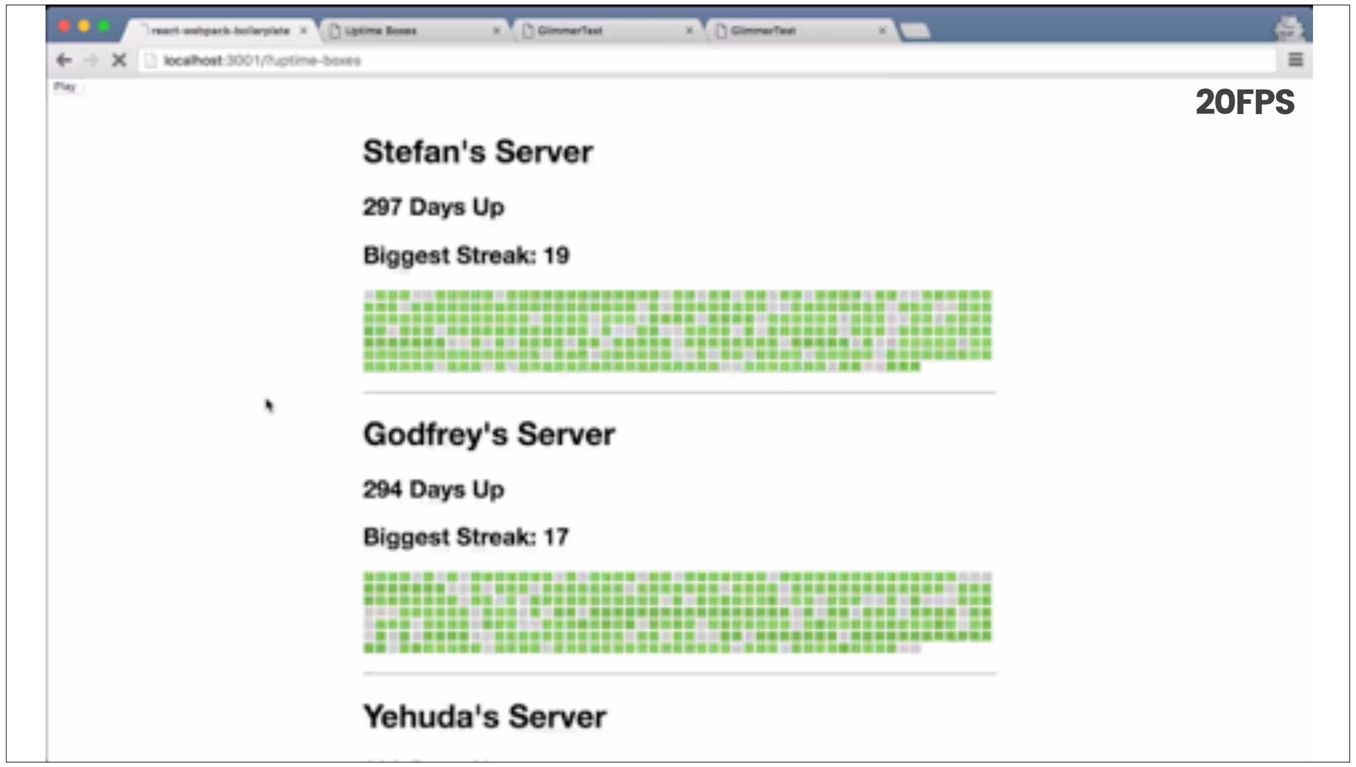


A few months ago, we made this new benchmark to guide our work on Glimmer 2.

There are over 1,000 components here on screen. Each box is its own component!

We kept the requirement to have stable DOM, so we kept the hover tooltip. We know React is pretty great at this stuff, and they're kind of the gold standard. What you're seeing here is the React version of this benchmark; the baseline. It's pretty fast, running at 20fps, trending a little down over time.

(this is React 0.14 in production mode)


20FPS

Stefan's Server

297 Days Up

Biggest Streak: 19

Godfrey's Server

294 Days Up

Biggest Streak: 17

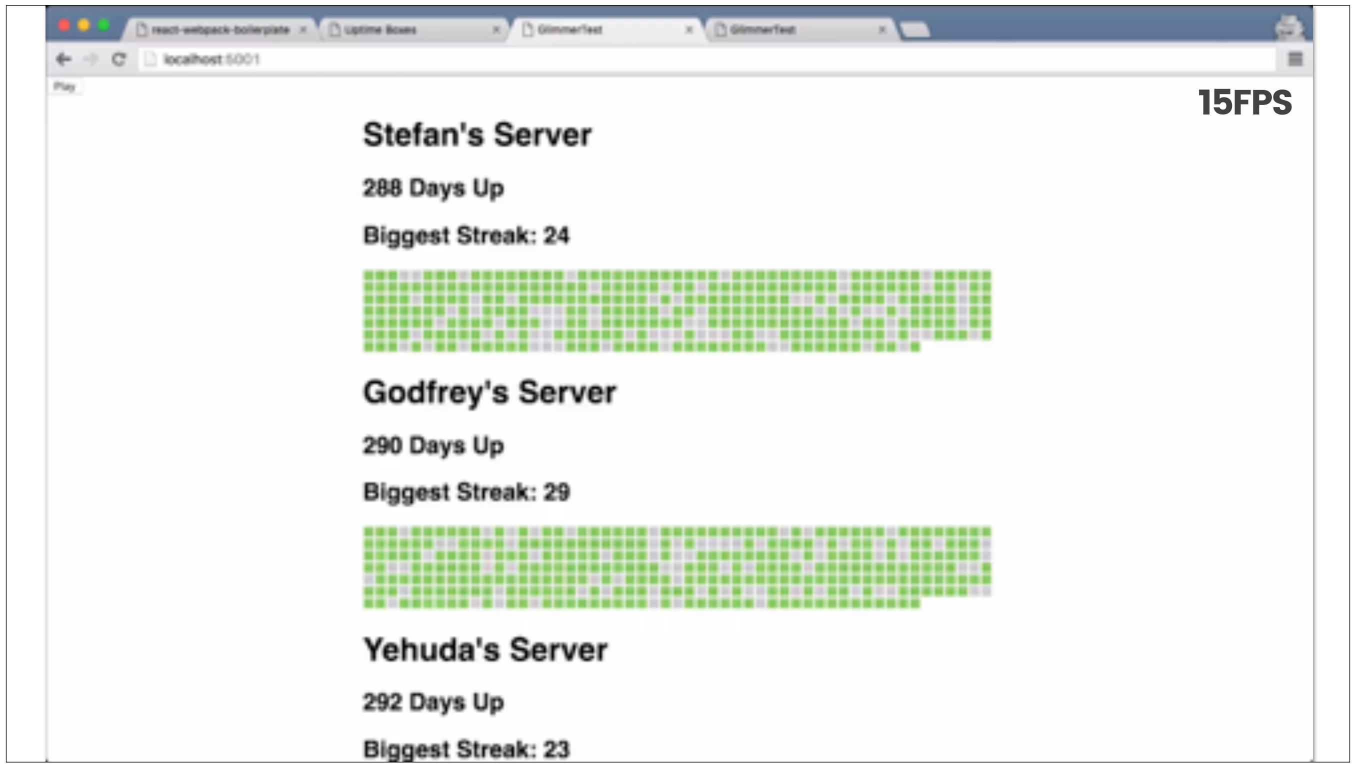
Yehuda's Server

A few months ago, we made this new benchmark to guide our work on Glimmer 2.

There are over 1,000 components here on screen. Each box is its own component!

We kept the requirement to have stable DOM, so we kept the hover tooltip. We know React is pretty great at this stuff, and they're kind of the gold standard. What you're seeing here is the React version of this benchmark; the baseline. It's pretty fast, running at 20fps, trending a little down over time.

(this is React 0.14 in production mode)


localhost:5001

Play

15FPS

Stefan's Server

288 Days Up

Biggest Streak: 24

Godfrey's Server

290 Days Up

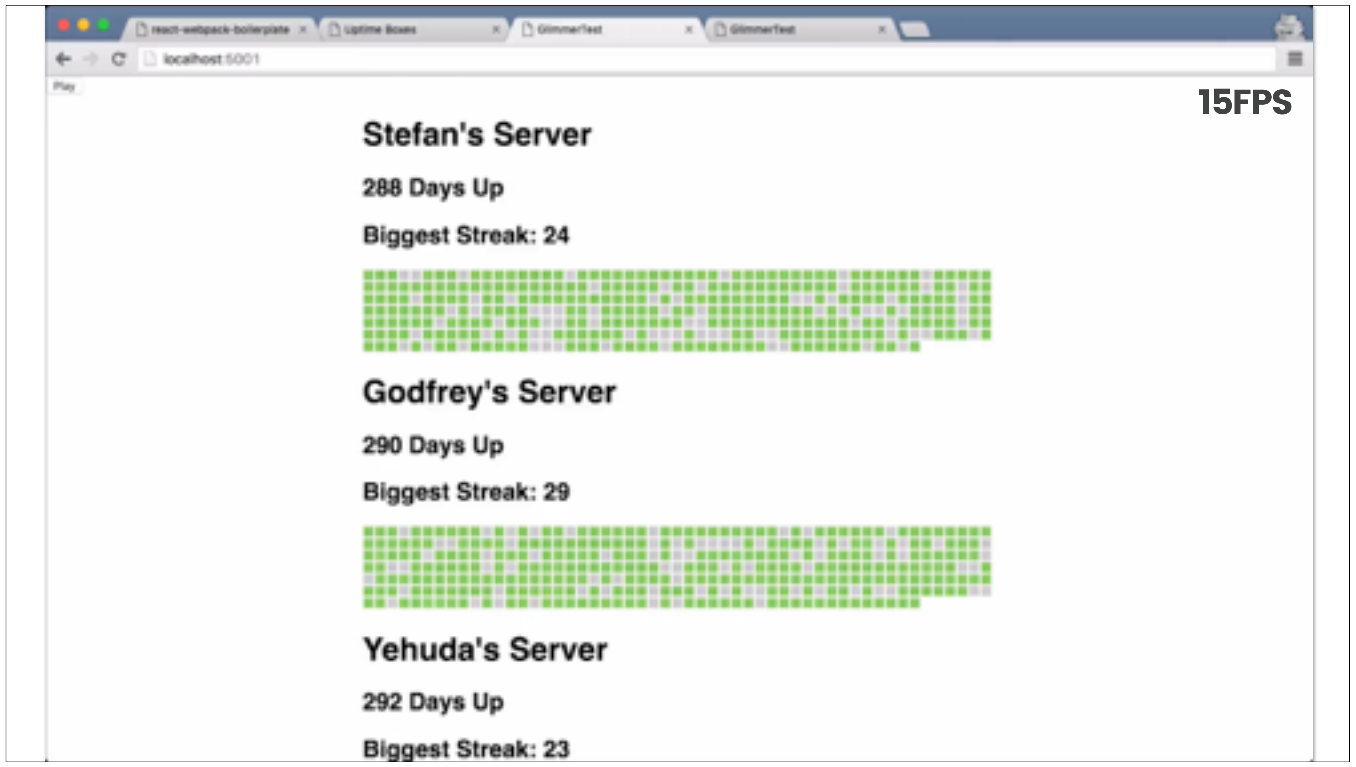
Biggest Streak: 29

Yehuda's Server

292 Days Up

Biggest Streak: 23

This is the same benchmark running on Ember Canary today with HTMLBars.
As you can see it’s 20-30% slower than React, running at around 15fps. (we got a reliable 15fps when running in a controlled environment and not recording the screen).


15FPS

Stefan's Server

288 Days Up

Biggest Streak: 24

Godfrey's Server

290 Days Up

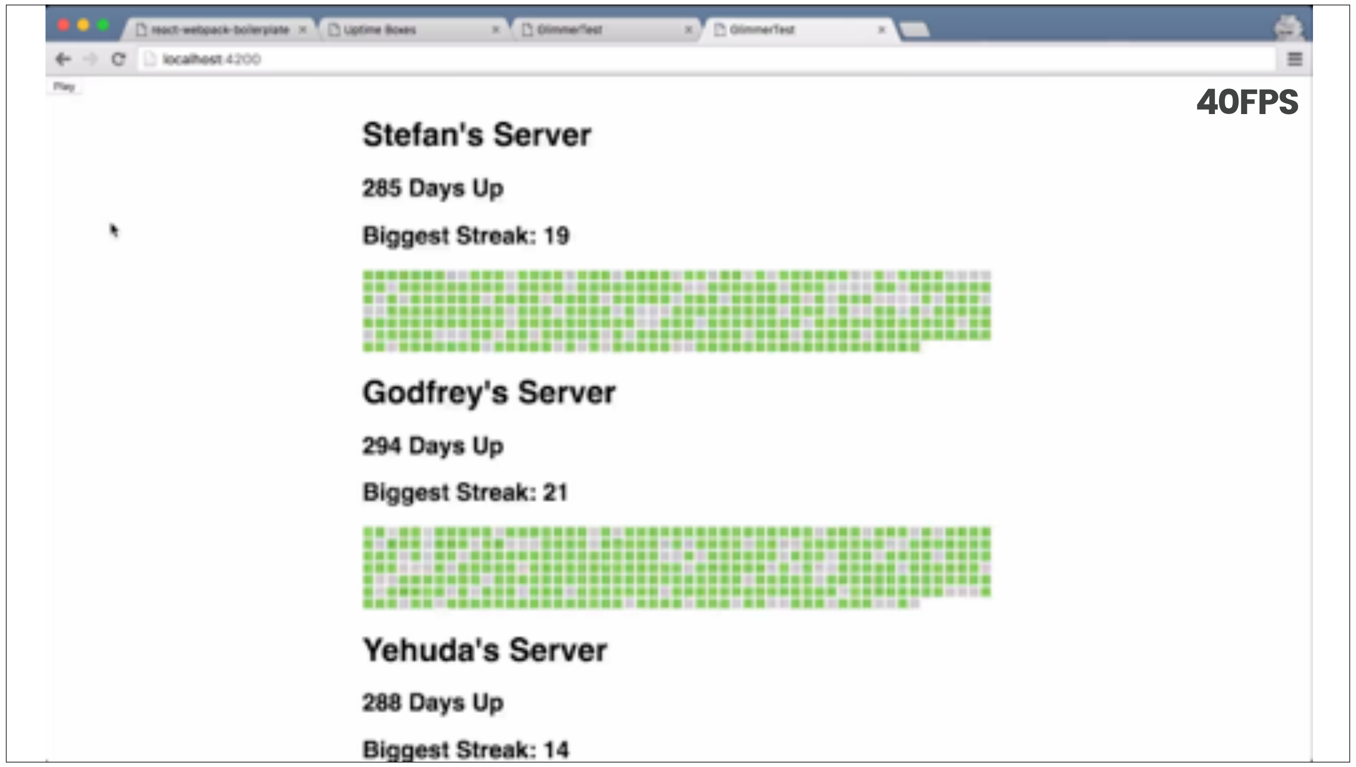
Biggest Streak: 29

Yehuda's Server

292 Days Up

Biggest Streak: 23

This is the same benchmark running on Ember Canary today with HTMLBars.
As you can see it’s 20-30% slower than React, running at around 15fps. (we got a reliable 15fps when running in a controlled environment and not recording the screen).



40FPS

Stefan's Server

285 Days Up

Biggest Streak: 19

Godfrey's Server

294 Days Up

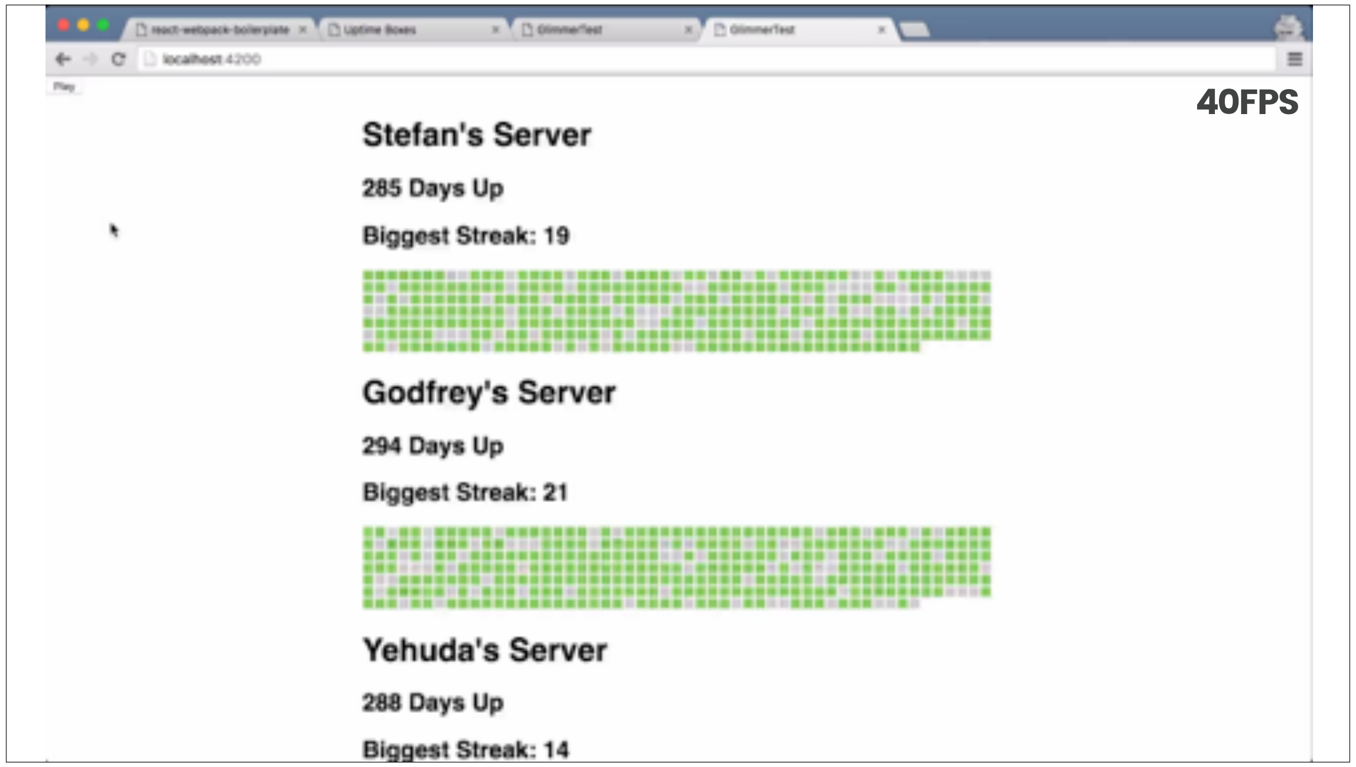
Biggest Streak: 21

Yehuda's Server

288 Days Up

Biggest Streak: 14

This is the same benchmark running on Ember Canary today with Glimmer 2.
As you can see, it is running at around 40 FPS with the same code, roughly twice as fast as the React version.



40FPS

Stefan's Server

285 Days Up

Biggest Streak: 19

Godfrey's Server

294 Days Up

Biggest Streak: 21

Yehuda's Server

288 Days Up

Biggest Streak: 14

This is the same benchmark running on Ember Canary today with Glimmer 2.
As you can see, it is running at around 40 FPS with the same code, roughly twice as fast as the React version.

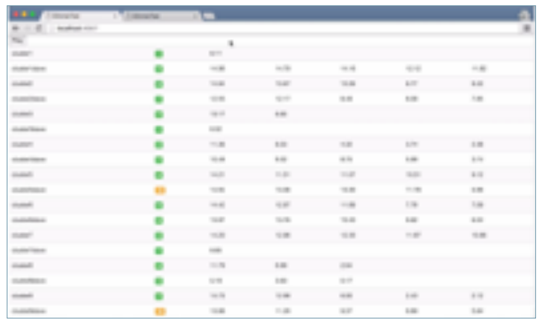
```
<div class="server-uptime">
  <h1>{{name}}</h1>
  <h2>{{upDays}} Days Up</h2>
  <h2>Biggest Streak: {{streak}}</h2>

  <div class="days">
    {{#each days key="number" as |day|}}
      {{uptime-day day=day}}
    {{/each}}
  </div>
</div>
```

templates/components/server-uptime.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  didReceiveAttrs() {
    this.set('upDays', this.computeUpDays());
    this.set('streak', this.computeStreak());
  },

  computeUpDays() {
    return this.days.reduce((upDays, day) => {
      return upDays += (day.up ? 1 : 0);
    }, 0);
  },

  computeStreak() {
    let [ max ] = this.days.reduce(([ max, streak ], day) => {
      if (day.up && streak + 1 > max) {
        return [streak + 1, streak + 1];
      } else if (day.up) {
        return [max, streak + 1];
      } else {
        return [max, 0];
      }
    }, [0, 0]);

    return max;
  }
});
```

app/components/server-uptime.js

This is a real Ember CLI app using `Ember.Component`, the router, etc. The exact same code runs on Ember 1.3, Ember 2.4 and Ember 2.6 Canary with the feature flag on.



DBMON

101 components

2x Speed Boost

to ~ 20

Stefan's Server

297 Days Up

Biggest Streak: 19

Godfrey's Server

294 Days Up

Biggest Streak: 17

Yehuda's Server

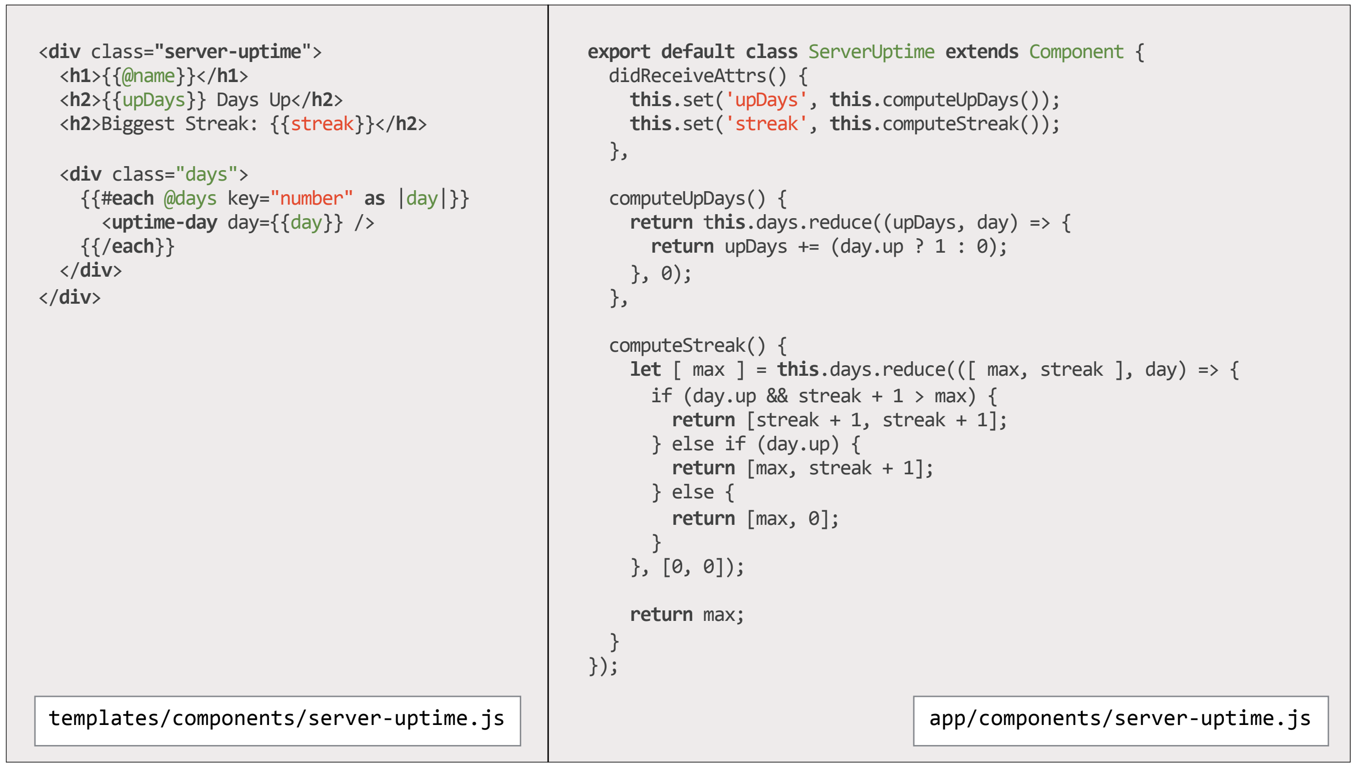
UPTIME BOXES

1,099 components

3x Speed Boost

to ~ 40

Our plan of baking components into the engine seems to be working – this benchmarks uses way more components than DBmon, yet it actually performs better.



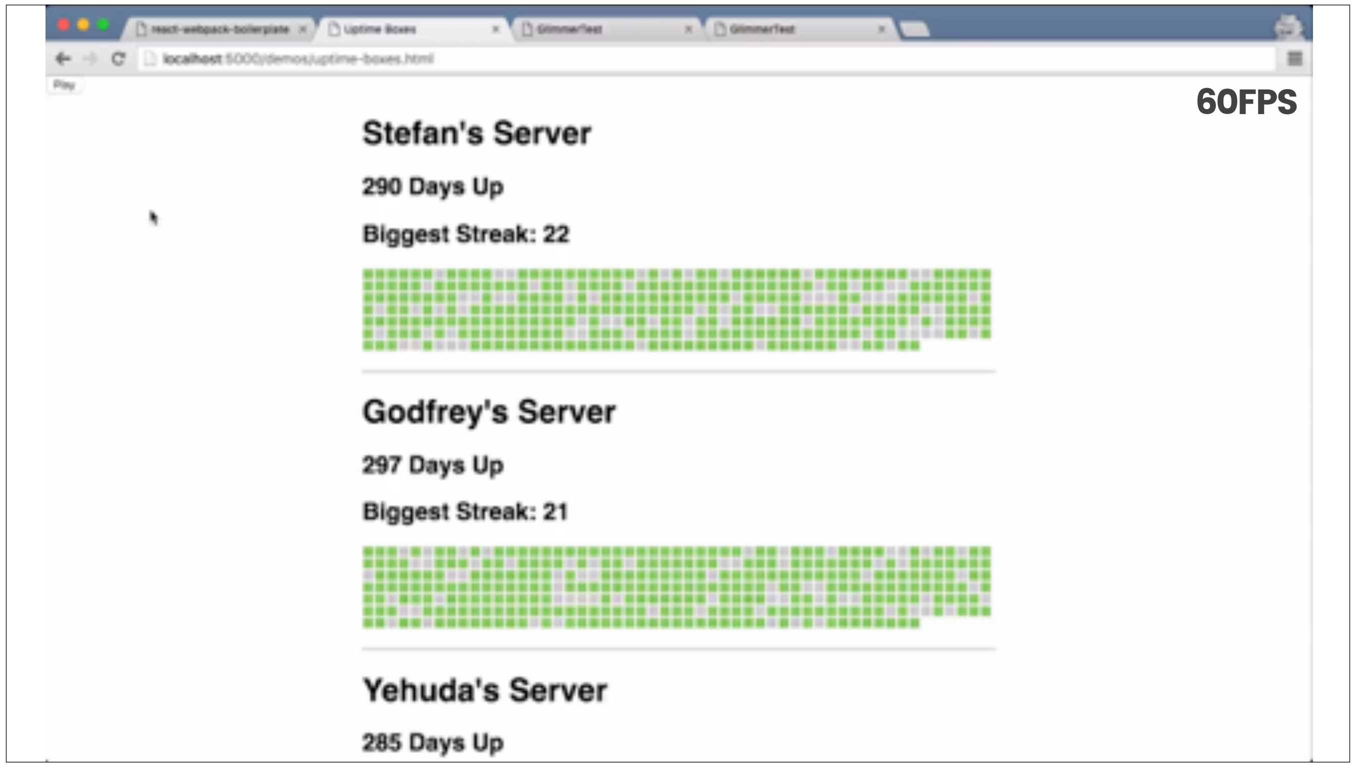
```
<div class="server-uptime">
  <h1>{{@name}}</h1>
  <h2>{{upDays}} Days Up</h2>
  <h2>Biggest Streak: {{streak}}</h2>

  <div class="days">
    {{#each @days key="number" as |day|}}
      <uptime-day day={{day}} />
    {{/each}}
  </div>
</div>
```

templates/components/server-uptime.js

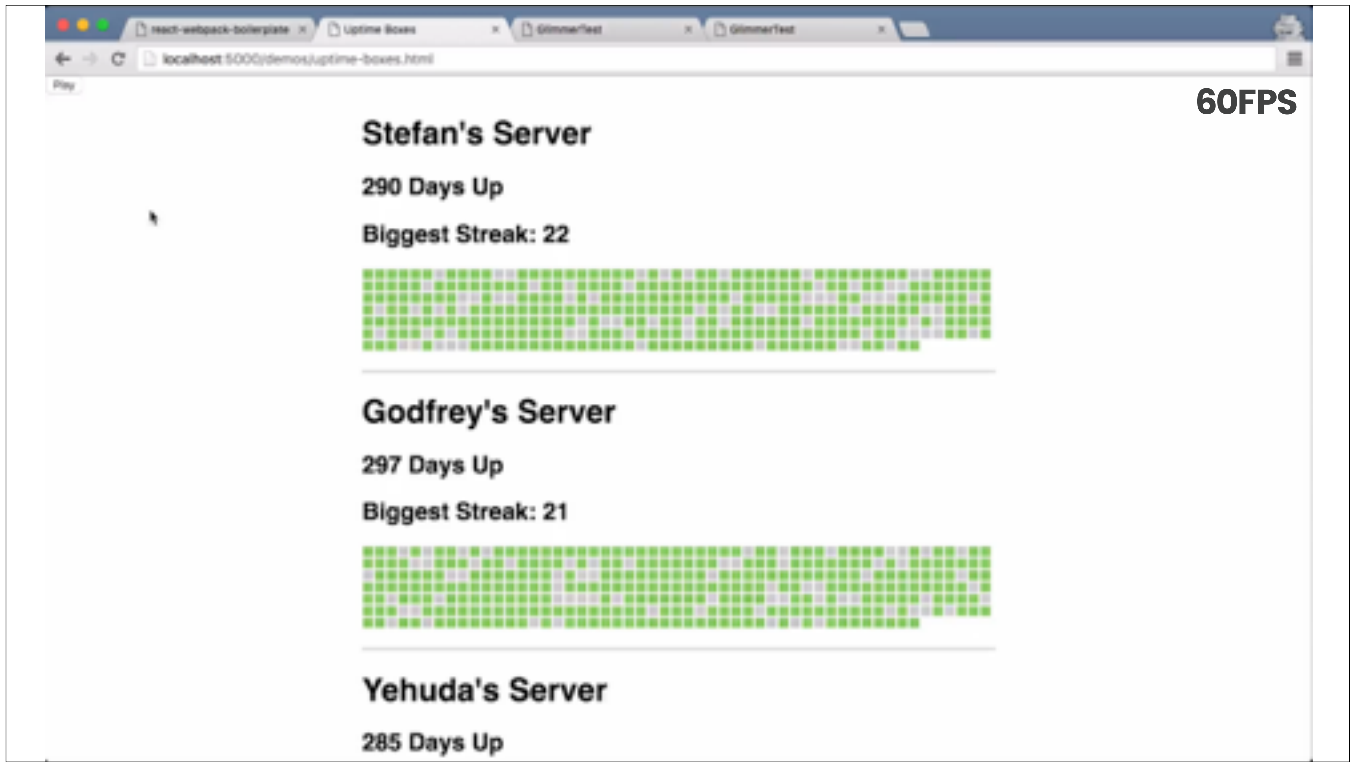
```
export default class ServerUptime extends Component {
  didReceiveAttrs() {
    this.set('upDays', this.computeUpDays());
    this.set('streak', this.computeStreak());
  },

  computeUpDays() {
    return this.days.reduce((upDays, day) => {
      return upDays += (day.up ? 1 : 0);
    }, 0);
  },

  computeStreak() {
    let [ max ] = this.days.reduce(([ max, streak ], day) => {
      if (day.up && streak + 1 > max) {
        return [streak + 1, streak + 1];
      } else if (day.up) {
        return [max, streak + 1];
      } else {
        return [max, 0];
      }
    }, [0, 0]);

    return max;
  }
});
```

app/components/server-uptime.js

But that’s not all! Let’s run the equivalent code on the Glimmer engine itself. As you can see, the Glimmer engine has absorbed a lot of the responsibilities of the Ember view layer.



The holy grail of 60FPS!
(It's actually faster than 60FPS (maybe 70FPS?), but we are using RAF so the browser eventually slows it down and caps it at 60FPS)



Stefan's Server

290 Days Up

Biggest Streak: 22

Godfrey's Server

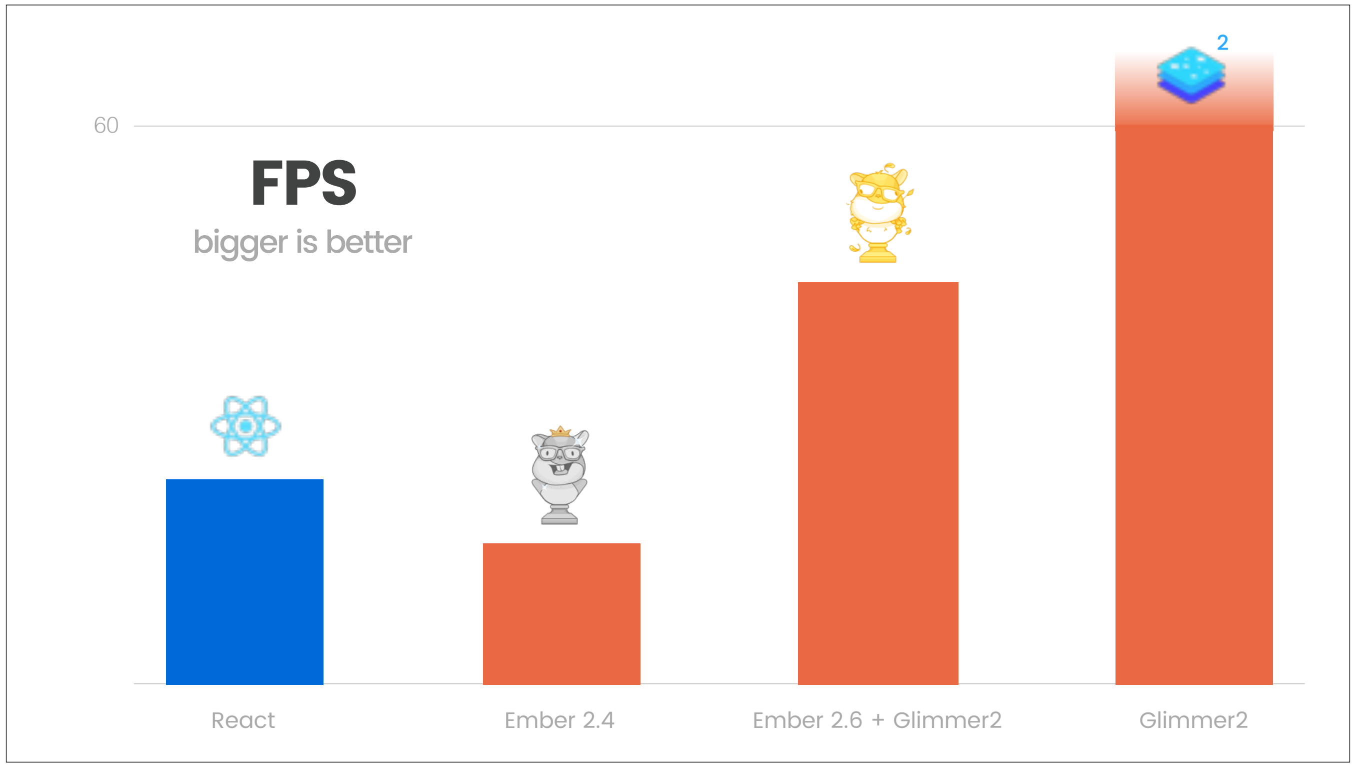
297 Days Up

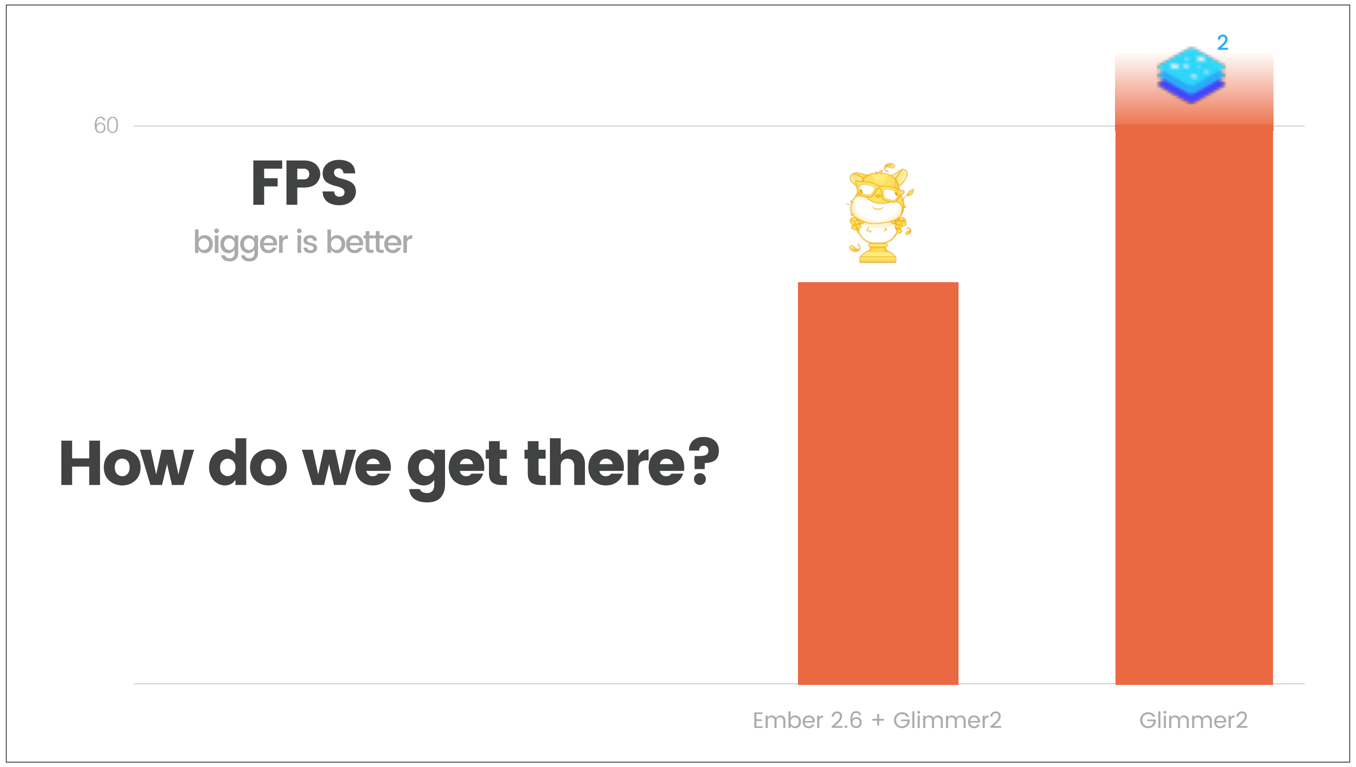
Biggest Streak: 21

Yehuda's Server

285 Days Up

The holy grail of 60FPS!
(It's actually faster than 60FPS (maybe 70FPS?), but we are using RAF so the browser eventually slows it down and caps it at 60FPS)

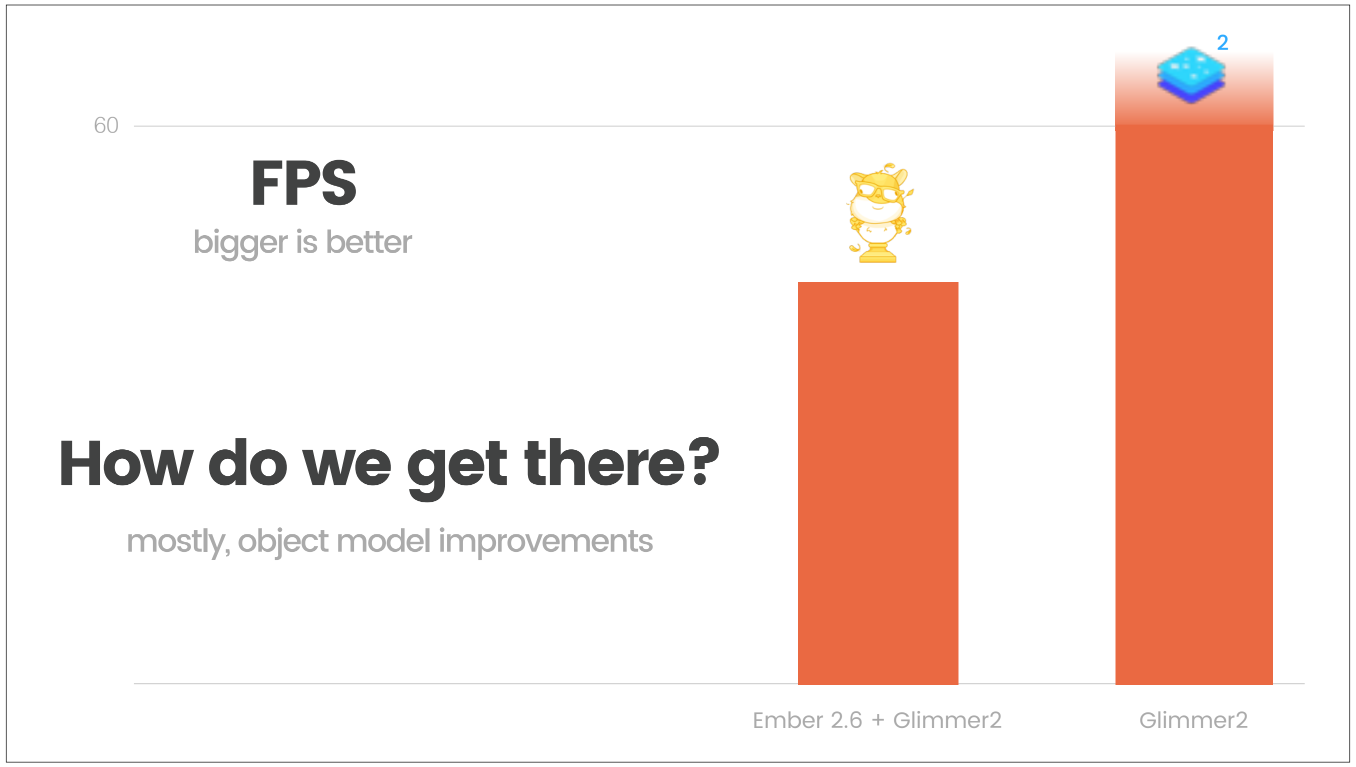

60
FPS
bigger is better
2
React
Ember 2.4
Ember 2.6 + Glimmer2
Glimmer2



There are still plenty of gains to be had here.

(click) What's the next frontier for Ember? How can we make Ember even faster, closer to Glimmer 2's performance?

(click) The most fruitful improvements are in the object model, to drive many of the learnings from the template engine through the entire framework. We hope to make progress here as we continue with integration.



FPS

bigger is better

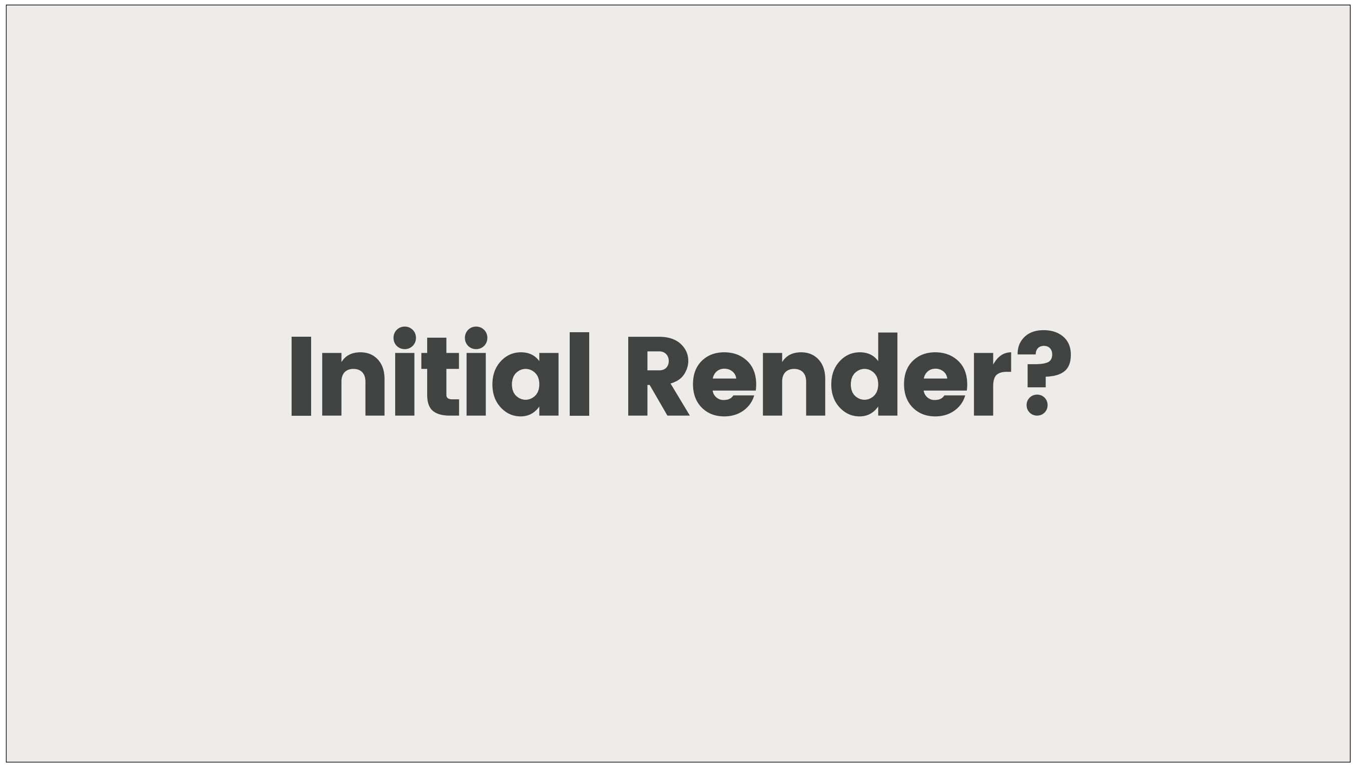
How do we get there?

mostly, object model improvements

There are still plenty of gains to be had here.

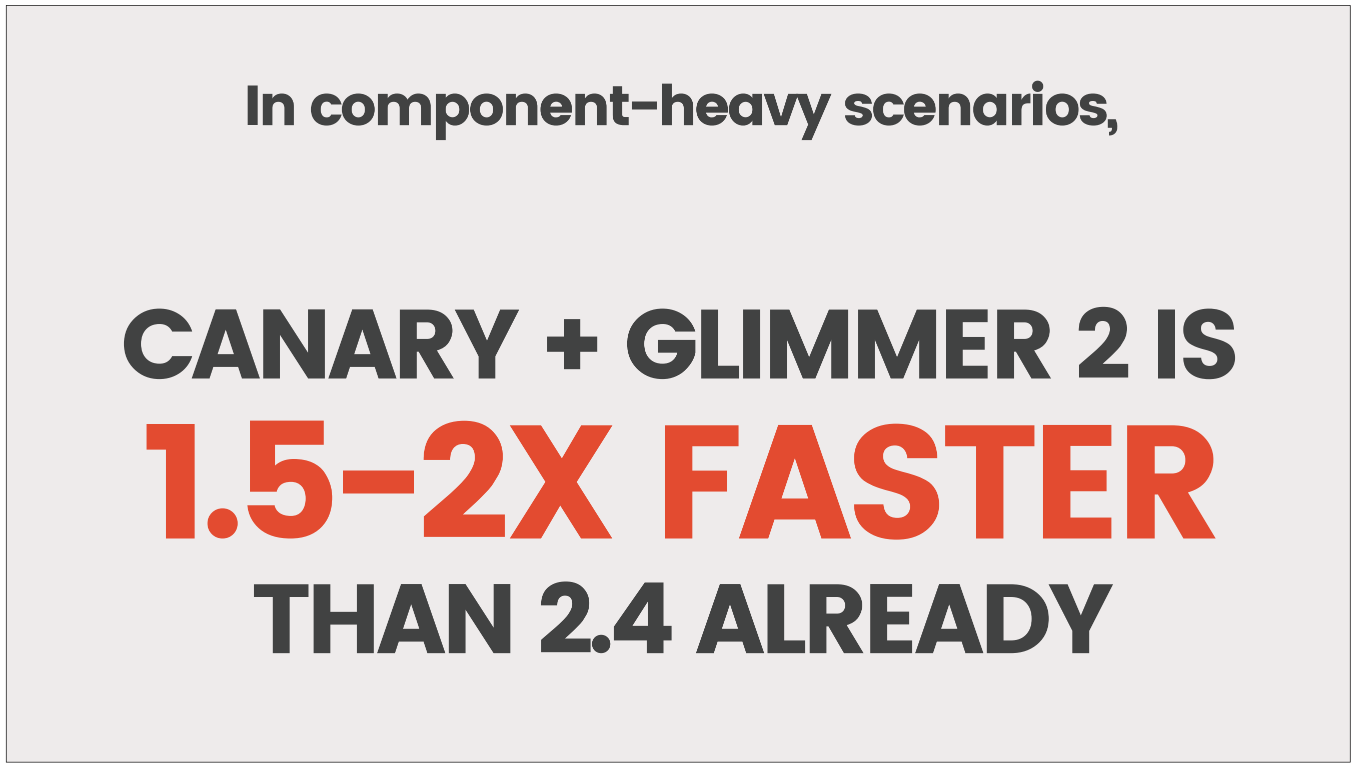
(click) What's the next frontier for Ember? How can we make Ember even faster, closer to Glimmer 2's performance?

(click) The most fruitful improvements are in the object model, to drive many of the learnings from the template engine through the entire framework. We hope to make progress here as we continue with integration.



Initial Render?

We haven’t forgotten about initial render performance; it’s just harder to make an exciting demo out of it.

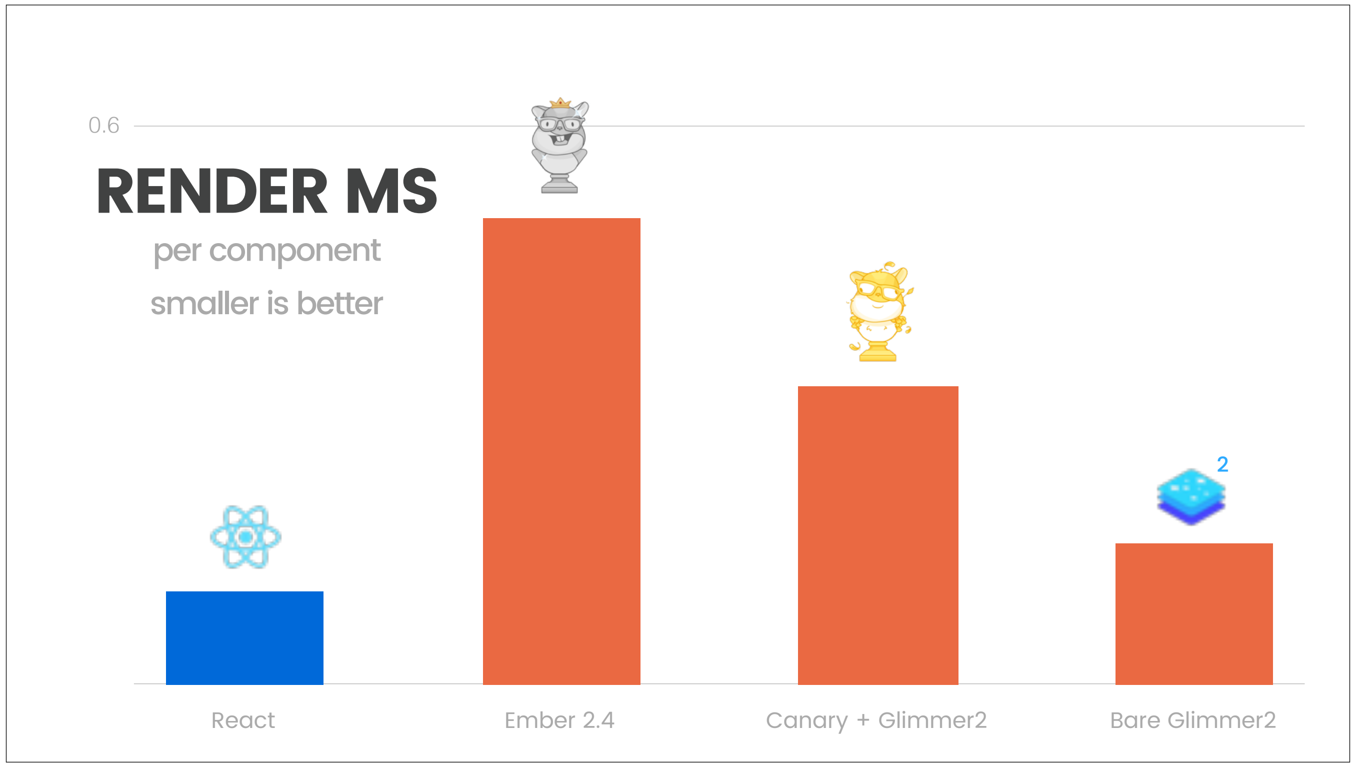

In component-heavy scenarios,

CANARY + GLIMMER 2 IS

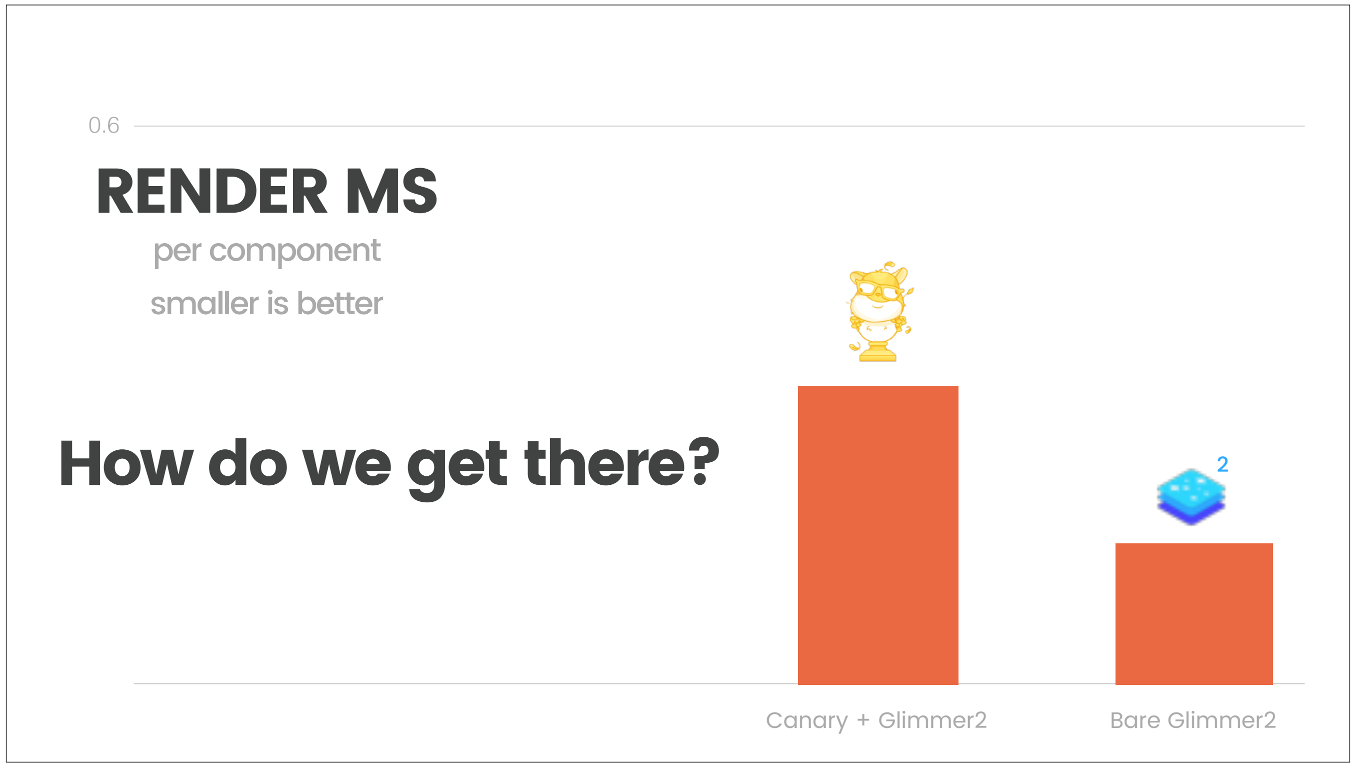
1.5–2X FASTER

THAN 2.4 ALREADY

on the component stress test.



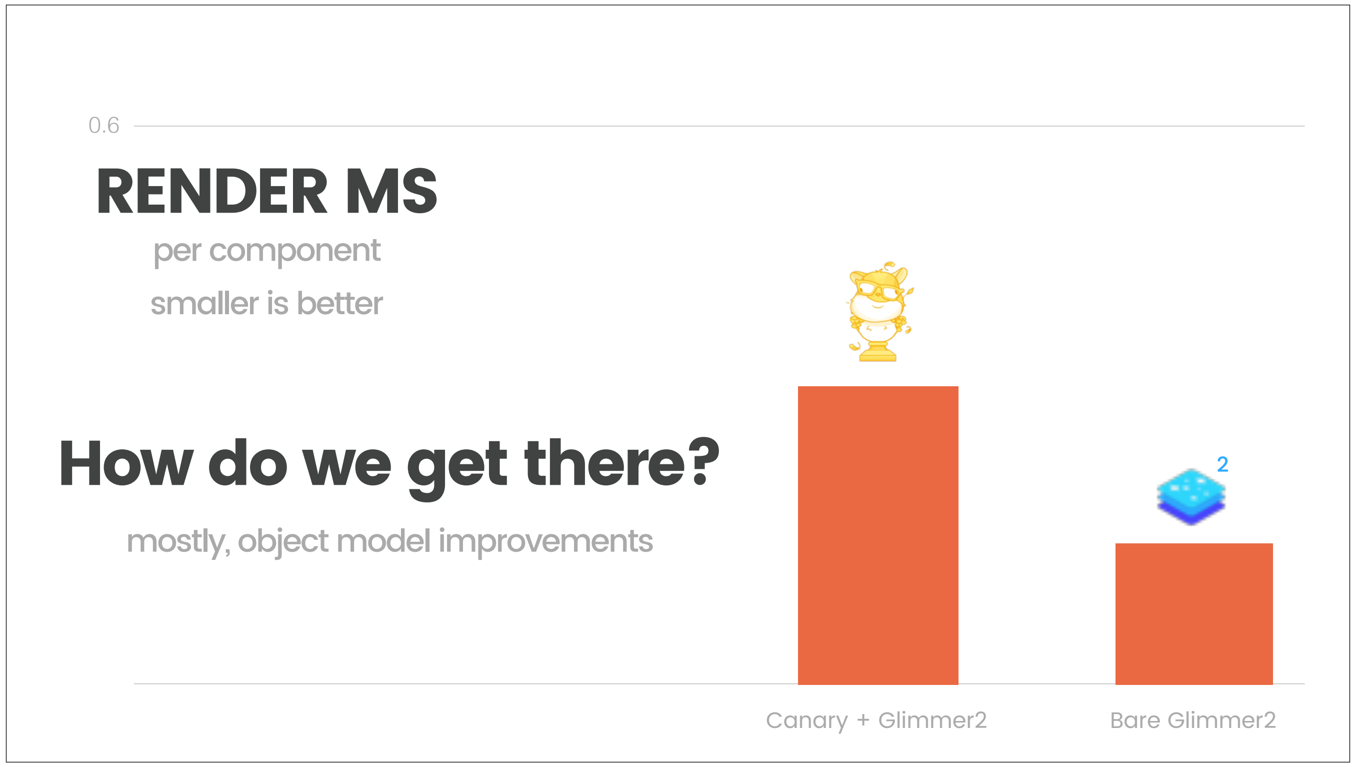
As you can see, bare Glimmer's rendering performance is already pretty close to React (and there's still more we can do).



How do we get there?

We're still bottlenecked here on object model performance, but we're hoping to catch up to get closer to bare Glimmer 2 performance, and eventually improve bare Glimmer 2 performance as well.

Note that this is just talking about wins in the render step. There are other wins as well…



RENDER MS

per component

smaller is better

How do we get there?

mostly, object model improvements

We're still bottlenecked here on object model performance, but we're hoping to catch up to get closer to bare Glimmer 2 performance, and eventually improve bare Glimmer 2 performance as well.

Note that this is just talking about wins in the render step. There are other wins as well…

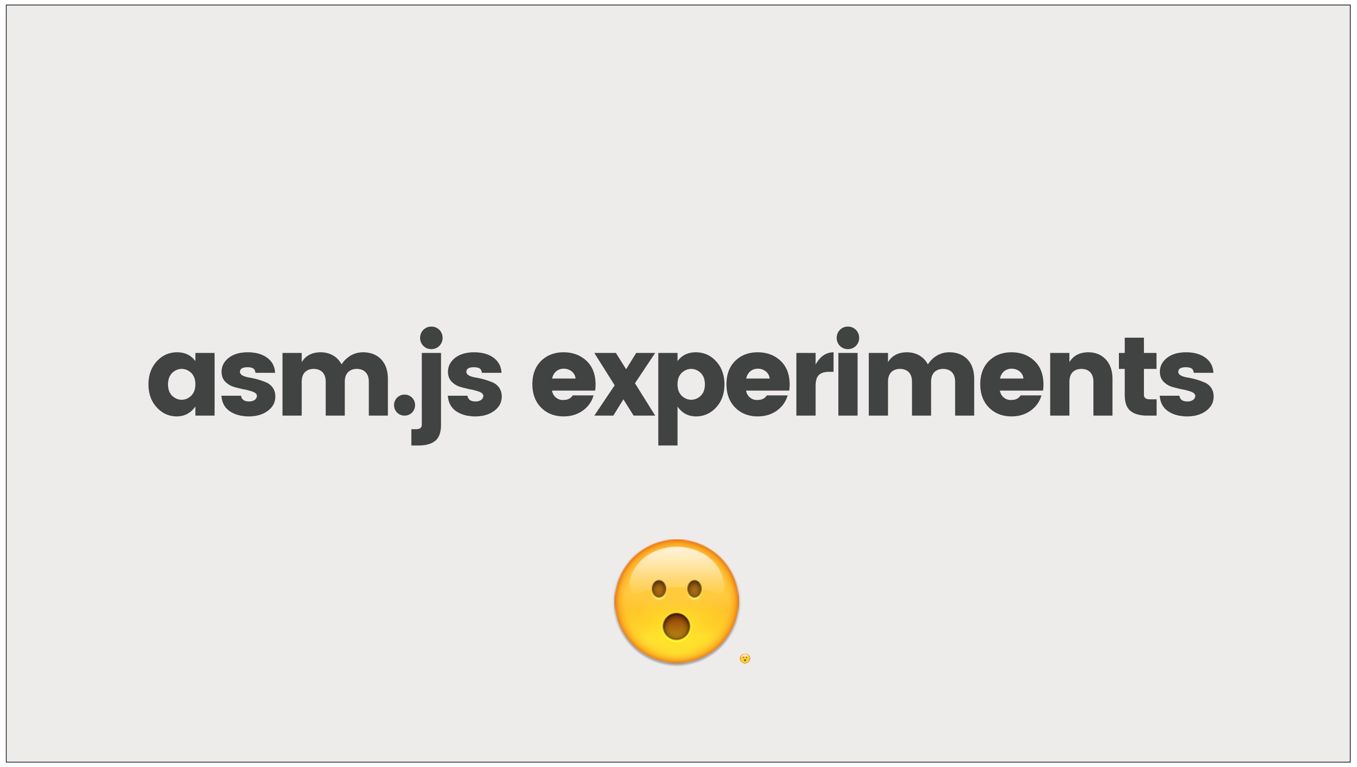
There are other wins:

GLIMMER 2 TEMPLATES ARE

5X SMALLER

THAN HTMLBARS

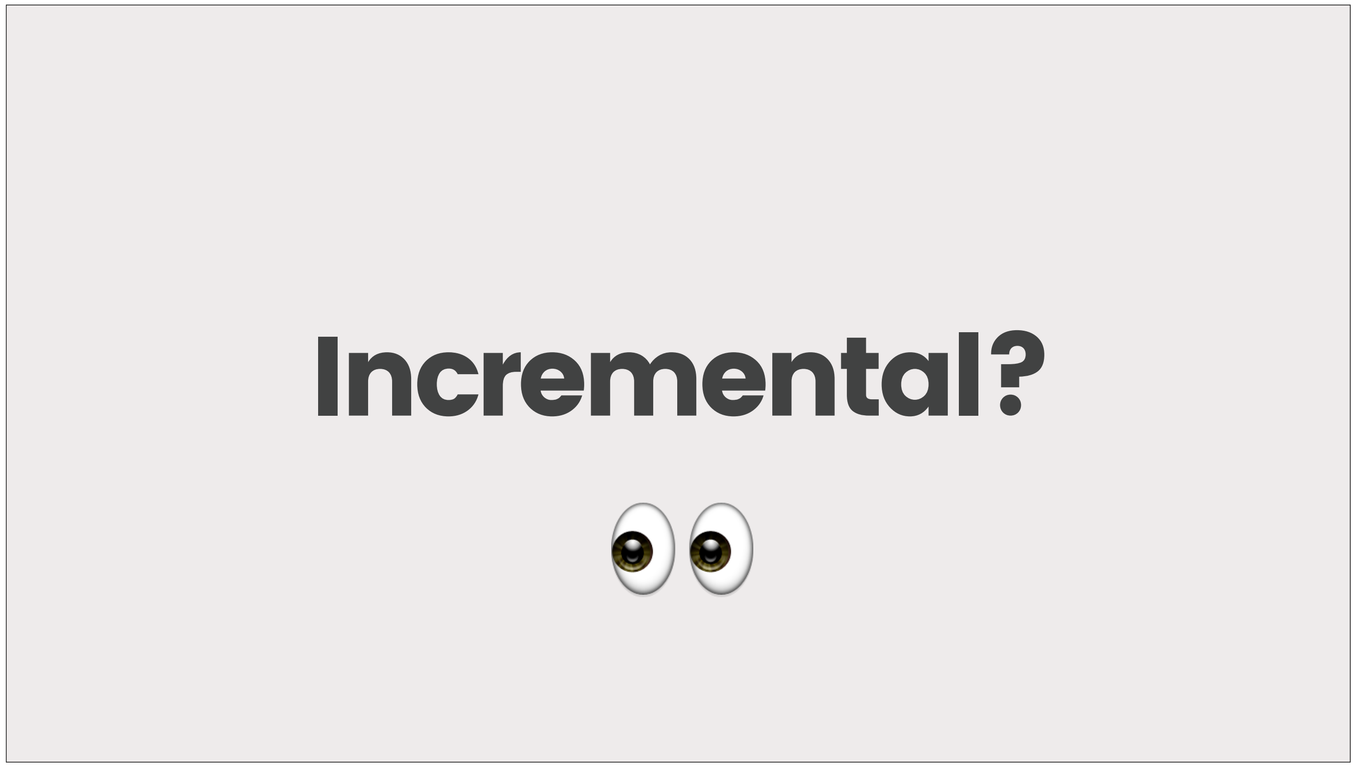
and they are lazily parsed rather than eagerly evaluated.



asm.js experiments

😮

We've been experimenting with moving some of the core algorithm to asm.js. There's nothing to report here yet, but we're working closely with the asm.js and wasm team and we feel optimistic.



Incremental?

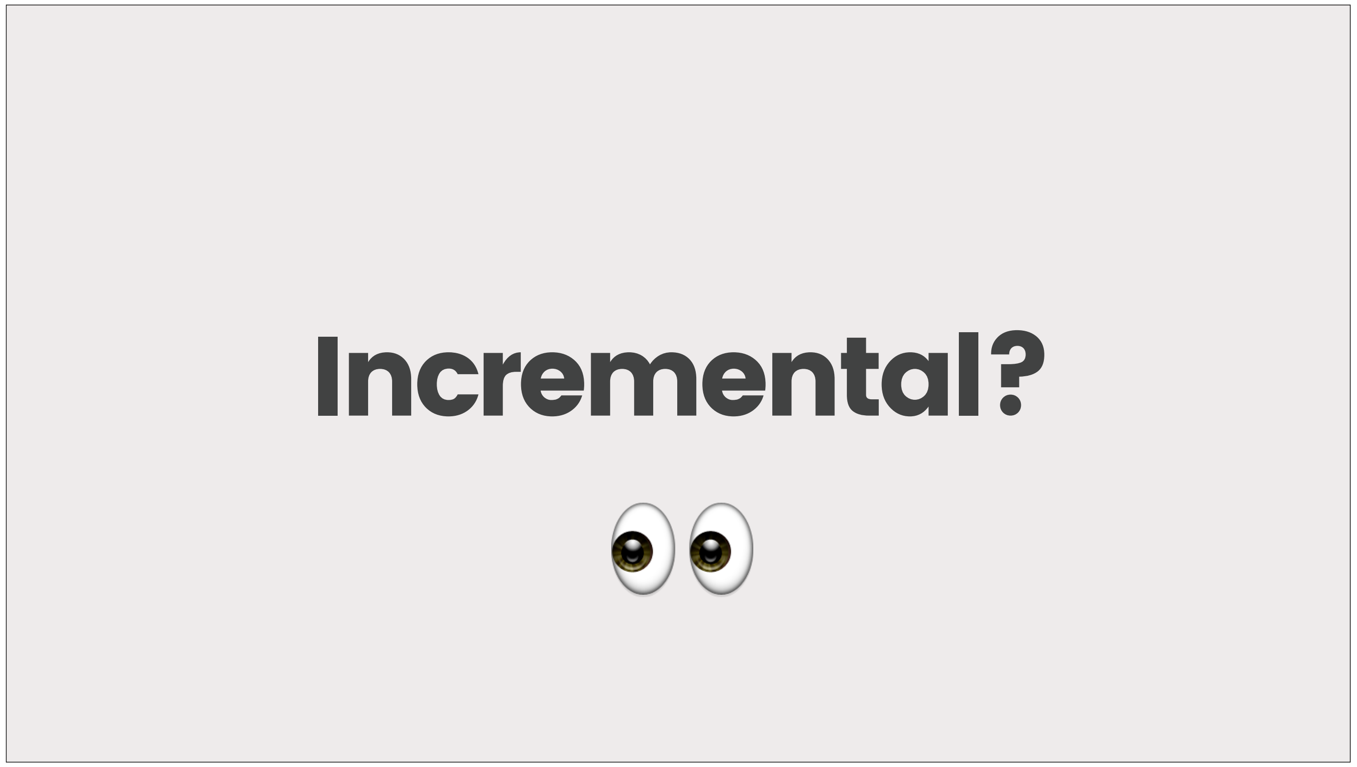
👀

So back to the question of how you manage something like Glimmer 2 in a fundamentally incremental process.

1. "Stability" at its core means not having to change your apps. Even when we do significant internal changes, we make sure not to break existing apps. The Glimmer 2 feature flag breaks no public APIs and introduces no new ones. So being careful to scope the change and not intermix other concerns is important.
2. Second of all, having a model for how to manage incremental change allows you to assess the risk of big projects like Glimmer and make sure you're not falling into a Perl 6 trap. In particular, we spend a lot of time thinking about how to make progress on integration before the whole enchilada is ready to ship.
3. We also lean hard on the process. While the first three months of Glimmer 2 were building the new core infra, we pivoted very quickly to figuring out how to build inside the Ember code base.

"Stability" means not having to change your apps. Glimmer 2 is classic innovation.

Talk a bit about the process, but also say that having a way to maintain progress while working, and a way to model incremental integration makes it possible to look at a project like Glimmer 2 and assess its chances of success.


Incremental?

👀

The first thing we did was just run the Glimmer tests in the Ember repo. Then we created a whole new package (ember-glimmer) and started putting tests in there that could run in both modes. The feature didn't work yet but we were able to avoid regressions in the part that did.

In short: having a good model for incremental development gives you a model for understanding the risk and limits of moonshots. And we've delivered moonshots to production far more predictably and to far more users than the unstable models.

Breakage

Ok so adding new features is nice, but what about breakage?

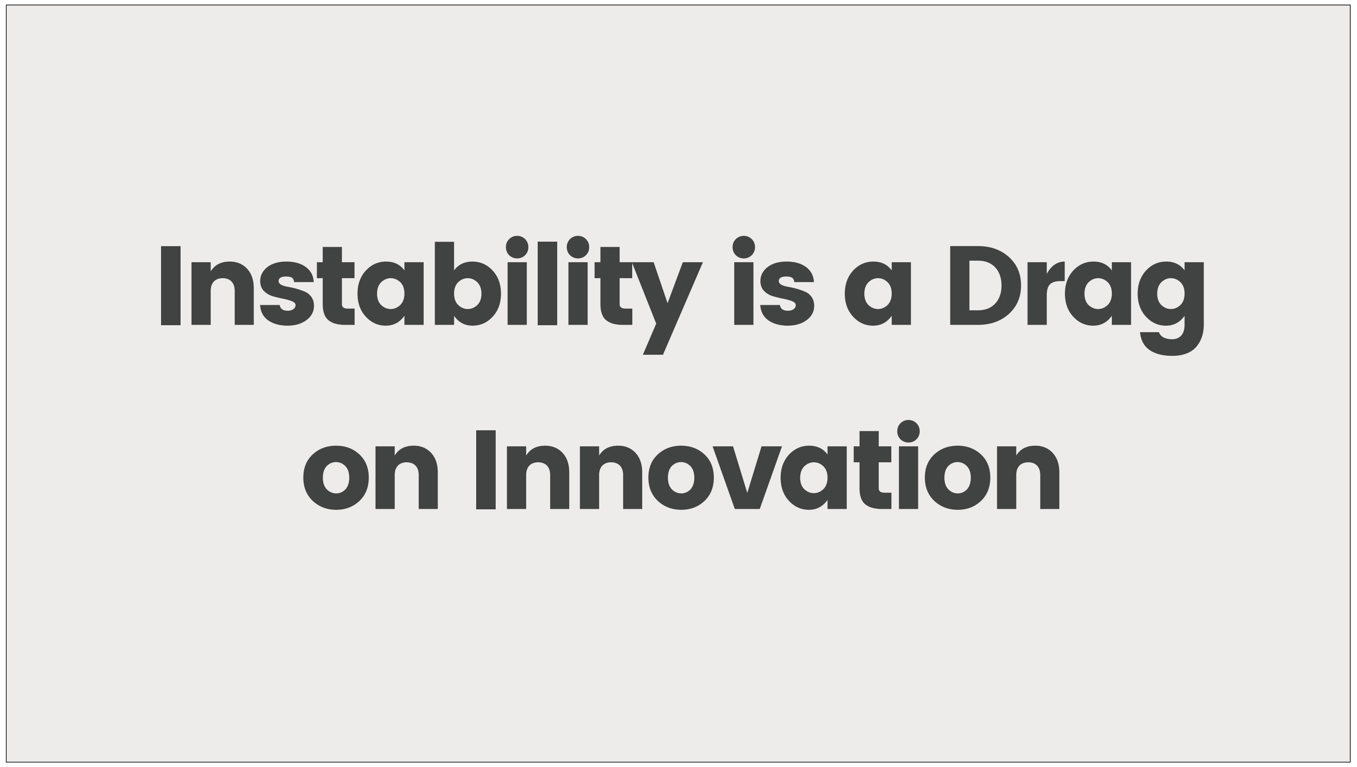
The Process of Breakage

- Semver (taken seriously)
- Deprecate to something
- Internal or external rewriting
- Deprecation Workflow
- Intimate APIs

1. We learned from 1.13. We won't do that again. In practice, stability tolerates very little breakage over a very long timescale.
2. taken seriously === we don't bump all the time; "just bump it; integers are free" ignores the cost to users and more importantly the ecosystem.
3. Svelte builds, part of the same RFC as LTS, allows users to remove deprecated features from their builds, to reduce the immediate need to delete them just to save bytes. This feature has become a more significant big-picture effort since the RFC, involving ES6 modules and tree shaking.

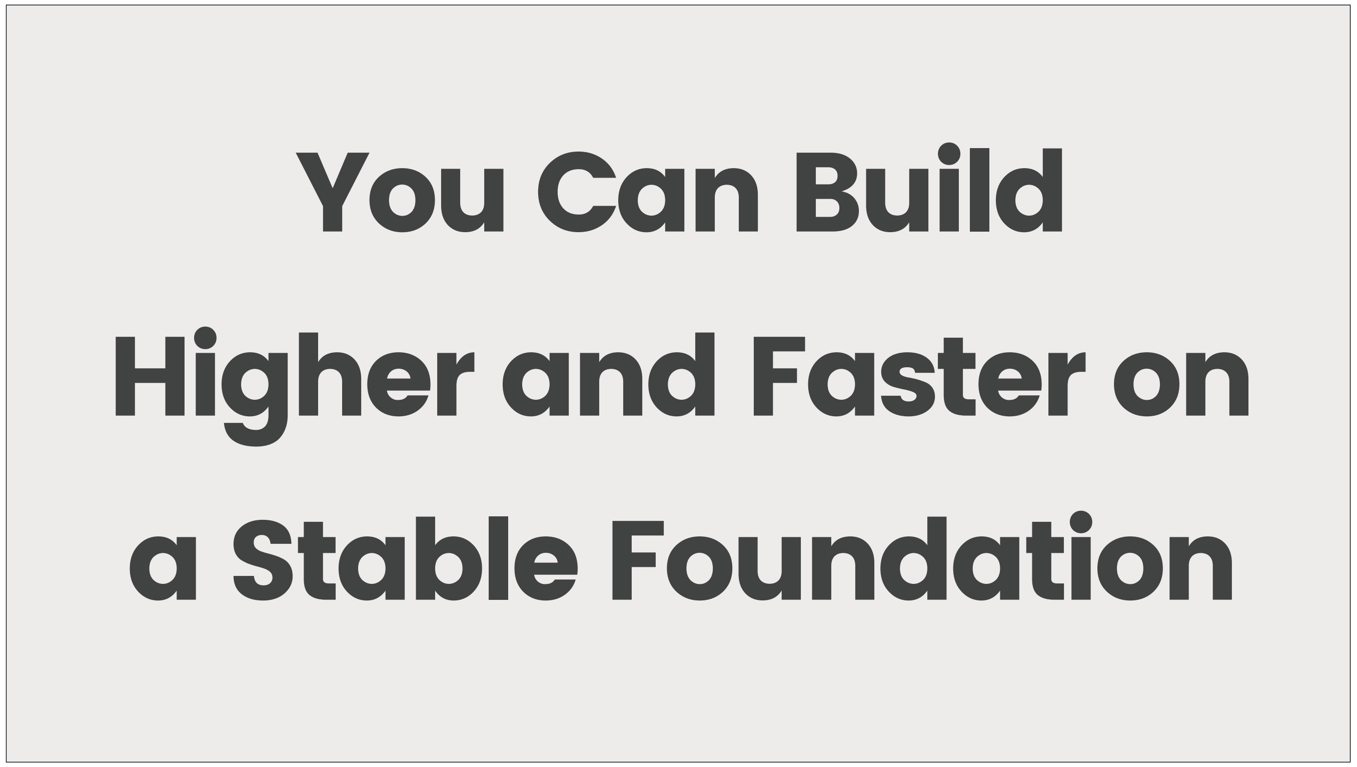
Stability without Stagnation

It's not just a nice slogan. It's about developing a process that helps you avoid getting stuck in a local maxima without wildly stabbing around in the dark looking for improvements (which creates ecosystem churn and breaks apps; it takes a while for an ecosystem to get humming; you can't afford to reboot that often, if at all).



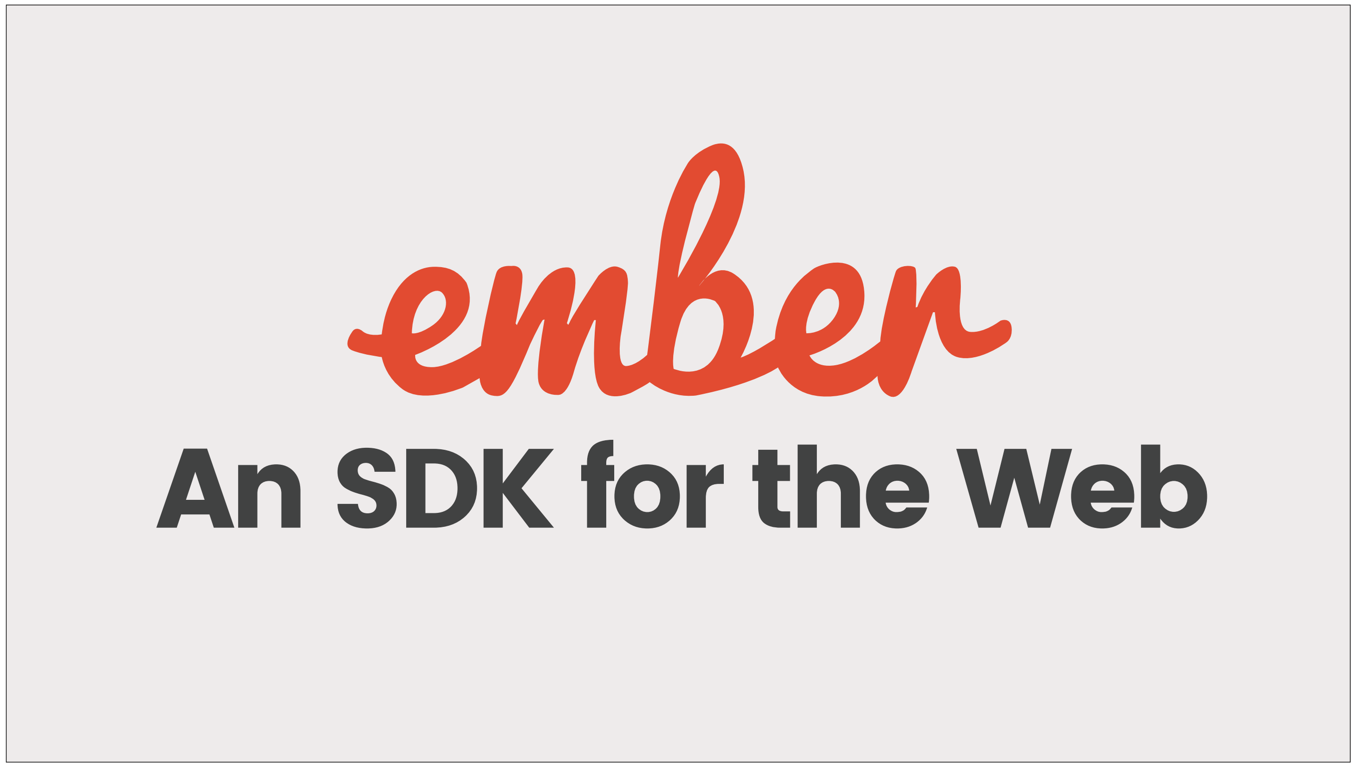
Instability is a Drag on Innovation

You've seen that there's a process for managing innovation without instability. But the alternative, that you must tolerate instability in order to innovate, counterproductively slows down total innovation across the ecosystem.



You Can Build Higher and Faster on a Stable Foundation

We should focus on building communities and ecosystems that allow people to innovate in their apps. I love working on Skylight, and I love taking off my framework hat and focusing all of my effort on how to best represent a user's name in our app. Let's love what we do.



ember

An SDK for the Web

To allow innovation in your apps, Ember takes on the responsibility of maintaining compatibility and nudging you towards better patterns and new functionality. We do this incrementally so that when you adopt new web technologies, you can upgrade to the next version, and the next version after that.

Thank You