# Engines	Details	Team Size
4	standalone routable Eager	2

The good

Decouple code into smaller more maintainable portions

The bad

scoped engines aren't supported yet

How we test

We don't *cough*

# Engines	Details	Team Size
3	standalone routable Lazily	100

The good

• Lazy-loading that "just works"
• Separate namespaces/module prefixes.
• The ability for logically separate parts of the application to fit our organization's structure.

The bad

• Managing the engine's resolver versus the dummy app's resolver in tests.
• Understanding how dependencies are included/deduped (the ability to include multiple versions of a dependency is a surprisingly common request).
• We plan on moving the majority of our 250k loc app into engines in a yarn workspace. I'd love to have a blueprint specific to this app so it's easy for another engineer to scaffold out their engine but I'm a little stuck on how to safely/sanely extend the existing engine blueprint.
• The documentation is out-of-date. I think I'd prefer that Engines docs be a part of Ember CLI docs so it's seen as a first-class citizen and there's fewer places to look.
• My personal pain point is not being able to write broccoli tests for engines, but ya'll have already seen my "memoize" PR. 😁

How we test

Mostly normal ember-qunit tests in the engine, plus a couple of acceptance tests in the host app.

# Engines	Details	Team Size
5	standalone routable Lazily	6

The good

Lazy loading, separation of concerns.

The bad

Unit/integration testing (new style). Keeping dependencies in sync (maybe yarn workflows are a solution for this)

How we test

In each engine, all types of tests.

# Engines	Details	Team Size
25	standalone both Lazily	2

The good

- Isolates the engines code from the host application.
- Can (with some manual wiring) discover and mount engines at runtime. Allowing the development of extendable web apps (our actual use case for engines)

The bad

None as yet, manual wiring for dynamic engine discovery is slightly inconvenient but to be honest understandable.

How we test

Not tested yet, due to dynamic discovery, a testing strategy is yet to be fleshed out.

# Engines	Details	Team Size
10	standalone routable Eager	15

The good

Core team support, lazy loading, isolation, concept of code splitting

The bad

- Not being able to access application queryParam and models with paramsFor and modelFor from inside engines routes (passing through services is a way, but we try to keep routing state out of services). Something like paramsForExternal/modelForExternal would help.
- Nested mounting of engines. I know it sounds dubious, but we have a structure where some engines have to be available in other engines. Didn't seem possible last time I tried. Also route names would become problematic I guess.

How we test

We don't write frontend tests currently

# Engines	Details	Team Size
1	standalone routable Lazily	1

The good

lazy loaded routes

The bad

testing

How we test

we don't as testing was painful. The engine is rather small and gets shipped untested. It's also not getting updated that often.

# Engines	Details	Team Size
1	in-repo routable Lazily	5

The good

Lazy loading/clear separation of concerns.

The bad

The styles leak in from the parent app.
Community support via addons is limited.

How we test

Don't yet (only started writing tests in the app)

# Engines	Details	Team Size
3	standalone routable Eager	40

The good

Fundamentally the isolation aspect is what makes engines great for us. It makes it easier to develop features independently, and it forces us to stop and think about where there truly needs to be crossover (e.g. when injecting a service from the host)

The bad

While the functionality is great, and obviously a lot of people are successfully using it, engines as a feature don't feel 'finished'. The 1.0 Roadmap writeup that Rob posted looks like it will cover most of our pain, but in particular:
- we've had to hack around a bit to get new-style testing working with our engines
- documentation feels somewhat piecemeal and dated
- engines are "just addons", except when they're not — they aren't first class in the tooling (e.g. `ember g component` still puts a reexport in `app`), but the degree of build-hacking that `ember-engines` has to do means they don't always interact with other addons the way a normal addon would
- lack of support for scoped engine names is a pain (our engines our internal packages where the addon name doesn't match the package name)

How we test

We have some degree of end-to-end Capybara testing, but largely we use the standard Ember testing setup with the engines' dummy apps.

# Engines	Details	Team Size
9	in-repo routable most eager, some lazily	5

The good

Isolation and encapsulation.

The bad

Testing; mostly for in-repo-engines.

The Ember Inspector is *really lacking* engine support. I know there are open GH issues about this but it's been a real challenge on-boarding new team members (Ember newbies) when the Inspector isn't very useful.

Also, the uncertainty around 1.0. I saw the post from @rwjblue the other day and this is great. Before that I couldn't help but wonder about our decision to adopt engines so broadly.

How we test

Poorly 😫.
We have some basic tests in the host application using the `engineResolverFor` method. This is one reason I'd like move our in-repo engines to be standalone so the test suite is also isolated.

# Engines	Details	Team Size
1	in-repo routable Lazily	3

The good

Allows speedup because the filesize of our bundle is reduced

The bad

We had one large monolith app, so we had to start splitting up our project with the use of addons. This slowed development a bit in the beginning, but ultimately improved as we improved our flow.

How we test

Currently we don't have any tests in place since it's a beta product of ours. We are planning on adding tests, but we still need to look at the best way to do this.

# Engines	Details	Team Size
2	standalone routable Lazily	11

The good

The idea behind it seems sound, but feels like might be better suited for OS shared projects than internal onesl ike we use

The bad

It actually makes the apps larger in our case, not smaller, since lack of tree shaking means we're pulling in a ton of stuff just for the engine a second time - feels like it should only pull in what the parent app isn't already providing.

It also makes dependency management HOORRRRRIIIBBBLEEE, we're constantly fighting dependencies when you try to npm link an engine to work on it locally. awful experience.

How we test

we test the engine itself in the engine code using mostly unit and simple acceptance tests, we have larger acceptance tests in the parent app

# Engines	Details	Team Size
3	both routable Lazily	12

The good

concern isolation

The bad

documentation is very spotty, inconsistent when discussing in-repo engines from normal engines.

How we test

Selenium based end-to-end integration testing.

# Engines	Details	Team Size
4	in-repo routable Some eager, most lazily	1

The good

Separation of concerns - ability to have standalone "packages"

The bad

deciding where the edge cases should go. Many similar or the same API calls with different use cases. Also, wish they could all be lazy loading

How we test

acceptance testing the complete application. Very difficult in this case to test separately

# Engines	Details	Team Size
2	in-repo routable Lazily	5

The good

Multi-stage builds (especially in delivering groups of functionality based on login access); strong boundaries between domains

The bad

This is less about engines and more about splitting up a large code base, but the team is a bit hesitant to move from in-repo engines to standalone because of the amount of PR churn that will be involved with our normal work.

The only other pain point about engines was the initial migration, which was mostly due to a lack of documentation. Our migration took place a year ago, so perhaps things are in a better place today.

How we test

Currently we use `preloadAssets` to preload all engines into our main app

# Engines	Details	Team Size
1	standalone routable Lazily	5

The good

Ability to use the same code across multiple host apps.

The bad

Just not enough resources and not enough information. Most guides and addons don't take engines into account, so there's a lot of trial and error to figure out what will work and what won't.

How we test

From the host app. No tests within engine.

# Engines	Details	Team Size
3	in-repo routable Lazily	2

The good

code splitting
lazy loading

The bad

impossibility to exclude some engines parts from build
(https://github.com/kellyselden/ember-cli-funnel)

How we test

acceptance testing of whole application

# Engines	Details	Team Size
1	standalone routable Eager	5

The good

Allows me to share common functionality between applications

The bad

Initial setup, some small bugs

How we test

Work in progress, most of the engine tests are still stuck in the main app.
Re above requestion, would like to transition to lazy loading but didn’t get to it yet

# Engines	Details	Team Size
1	in-repo routable Lazily	30

The good

It helps to isolate code, separate logic, make app more lightweight

The bad

not much

How we test

I test them as normal tests in ember-cli

# Engines	Details	Team Size
3	both routable Lazily	4

The good

* Clear separation of concerns
* The possibility to develop code separately
* The possibility to use the engine also in a different project
* Lazy-Loading

The bad

* Lack of IDE support
* Complicated setup of the "interface" between the engine and the host app
* Sometimes it's inconvenient to develop an engine if you have to change things in the host app and the engine (especially if it's not an in-repo engine). It often leads to some workflow as follows: "Change the code of the engine, commit, push. Update package.json in host app, run npm install, check if changes in engine work in host app, if not start the cycle again :-D "

How we test

We use the stuff which is provided by Ember-CLI

# Engines	Details	Team Size
1	standalone routable Eager	4

The good

Isolated, shareable concerns

The bad

Using Ember Data's store, having engine-specific models

How we test

Standard tests for the engine, acceptance test in the consuming app

# Engines	Details	Team Size
3	standalone routable Lazily	6

The good

Enables logical breaks in code responsibility. (ie Admin, Client, Reporting, etc)

The bad

Dependencies are not isolated

How we test

Tests are run individually in each Engine and on a lesser level from the consuming application

# Engines	Details	Team Size
14	in-repo routable Lazily	4

The good

Code splitting was the original reason.

Nowadays, I really like how we can bring in a new dev, give them an engine, and let them rip. They can use their own project structure, CSS style, JS style. (for better or worse). We can wall off a new devs code style from our main product. We are a super busy startup and honestly we don't have time to check everything or put our stink on every single line of every file. Engines gives us the ability to hire an experienced dev and say. "Write tests for your stuff, we will write acceptance tests from the main app, otherwise... dont be a ding dong and try to do a good job"

Another thing I like is that we can slice entire off entire parts of the application by simply commenting out the 'mount' point in the router.

The bad

The only big one was figuring out how to share things (we eventually landed on a shared in-repo addon that all the engines share.) Things like scss vars, common components, mixins, helpers etc. We wanted to be able to share some things between the main app and the engines easily. It was hard to set this up, mostly because we implemented engines before yarn workspace support, so we had to do a ton of sym-linking by hand.

The second thing that is slightly annoying is the `link-to-external`. Because we share some components in a parent in-repo-addon we have to use `link-to-external` everywhere in that addon, just in case the component is used from the engine. Would love if `link-to`'s just worked.

Another big one for our project is that a lazy engines css is lazy loaded... even when this page is rendered in fastboot. This sucks because that means all of our engines have an annoying FOUC when visiting them in a fastbooted ember app.
I have been trying to bring attention to this here:
https://github.com/ember-fastboot/ember-cli-fastboot/issues/590

The only other thing that I see our juniors struggle with is the engines config. I wish there was a documented way to share config values. But yeah.. its not that hard to do... just confuses our juniors.

How we test

Same way we test our ember app. We colocate our component integration / unit tests. Then we have a ton of high level acceptance tests in the main app.

# Engines	Details	Team Size
4	in-repo routable Lazily	3

The good

Business model splitted with code in engines, lazy loading, logic separation and sharing services

The bad

Last time I checked, fastboot didn't work with engines

How we test

I'm sorry, I'm not using any tests from ember, just rushing into mvps

# Engines	Details	Team Size
4	in-repo routable Lazily	10

The good

- splitting a large app into lazy-loaded bundles
- the strict encapsulation can help encourage good architectural design

The bad

- addons sometimes do fancy things assuming specific things about their parent, then break when used in engines
- including stuff can be confusing, e.g. importing assets from node_modules in lazy-loaded engines
- weird interactions using in-repo addons to share components/logic among in-repo engines -- third-party addons used by our in-repo addons get confused
- most of our engines don't conceptually make sense as standalone, since they're used just once by our app, but I'm pondering whether a monorepo of standalone engines would be cleaner...

How we test

Still figuring out a good testing story...

# Engines	Details	Team Size
7	in-repo routable Lazily	2

The good

I want to organize code better and have code splitting (our app is very big and loading times just to get vendor/app.js is long). Also in one (less important) situation (admin interface) have code not given to a user unless they have access.

The bad

- services sharing
- external routing sharing
- needing to refactor components/helpers into add-ons to use in both
- `ember g route` doesn't add to `routes.js`
- hacks needed to get guest/authed redirects working
- builds are slower
- weird infinite loop errors if there is an error in an engine
- bunch of boiletplate I need to hookup just to get a new engine working
- have to restart `ember serve` if I change `package.json` on the engine
- lack of direction in general from the team, feels like engines should be shelved once module unification and code splitting features are done

How we test

We don't.

# Engines	Details	Team Size
0	in-repo routable Lazily	4

The good

Async code; composable apps; faster load times;

The bad

Setup with existing projects; overhead, boilerplate;

How we test

We don't because we haven't tried it yet

# Engines	Details	Team Size
1	standalone routable Lazily	4

The good

Black-block isolation. Other than build issues / package issues I am about to describe they just worked(tm) which was fantastic. I would rather have build / package issues than have the engine break while running in a production environment.

The bad

Build pipeline. Dependencies. Depending on an addon which depends on an addon which depends on a raw package.

How we test

Engine package contains rendering/unit level tests. Engine package lives in a multi-package repo with a full-fledged Ember application which exists solely for hosting the engine for application tests and development. This structure was created so we could run application tests against multiple engines in unison in the future.

# Engines	Details	Team Size
8	standalone routable Eager	8

The good

It's far easier to develop a separate portion of your UI, without having to involve everyone. Also easier to isolate / version / roll back when issues arise.

The bad

Dependency management is a nightmare. We discover all kinds of conflicts - routes, components, services, etc that conflict with the mounting app. And when we want to update a shared dependency it's a very involved process (many engines share the dependency). We rely heavily on experimentation when we have to do broccoli / build process stuff as there isn't a ton of engine-specific info - much assumes you're in either the consuming app or (less frequently) an addon. Using engines has unfortunately also slowed the deploy process as well - when we iframed the client in, we could push at will, but publishing as part of a the mounting app binds it to our main app's release cadence

How we test

Unit, Integration, acceptance. Selenium functional tests against prod, testing, dev environments

# Engines	Details	Team Size
1	in-repo routable Eager	12

The good

I love that it allows us to architect our app in a way that aligns with our team structure so that independent teams can focus on the parts of the app they own without needing to worry about causing problems for others.

The bad

Sharing core functionality like models has pushed us to move a large part of our codebase into Addons. I'm pretty sure this is a good thing for us, but it definitely hampered development speed as we adjusted.

How we test

Unit and Selenium tests

# Engines	Details	Team Size
4	standalone routable Lazily	8

The good

The ability to prevent large monolithic applications.

The bad

Lack of documentation and examples.

How we test

Isolation test for each engine and cucumber.js for the whole app.

# Engines	Details	Team Size
32	in-repo both Some eager, most lazily	200

The good

Code-splitting for lazy loading

The bad

Code-splitting for lazy loading.
Fails to de-duplicate transitive addons that are shared by host and engines' dependencies.

How we test

Along with the host tests... one big, slow building, slow running glob. Currently trying to build & test engines separately

# Engines	Details	Team Size
5	standalone routable Eager	5

The good

The micro front end is a clean way to pair UI with micro services.

The bad

These are not exactly pain but would be nice to have.

1. Engine export namespace
Occasionally we want to export some module from Engine to app. One example is the use of shared ember-data store. And we would want to export model/adapter/serializer from engine to app. To make sure the exported modules won't conflict with anything else existing in the wrapper UI or is being exported from another engine, we have to name these modules with a prefix. Something like `models/engine1-user`, `adapter/engine1-user`. Then we have to use this verbose type name when we're interacting with any ember-data API (this.get('store').query('engine1-user')) which is kind of annoyed. It would be nice if we can provide an export namespace to individual engine so these exported modules don't collide easily.

2. build performance
We are using FROST as base components library which includes around 20 ember addons. Each engine we're building has a large set of duplicated dependencies(frost components). When we install all 5 engines into the wrapper app(all the app do is mount engines), the build time of the app is around 20min given the amount of the duplicated deps it has to build. We have to customize ember-cli to speed up the build process at the moment. It would be nice if the build process can be improved given the context of ember-engine.

How we test

Acceptance tests in the dummy app of each engine. Integration and Unit tests as usual.

# Engines	Details	Team Size
1	standalone routable Eager	10

The good

Code reuse

The bad

Tight coupling. And the incomplete testing story

How we test

Thats the problem

# Engines	Details	Team Size
4	both routable Lazily	15

The good

Isolation between multiple apps and code splitting.

The bad

Testing would be the biggest one. Also, engines not having their own router and how global `didTransitions` work with engines and the consuming app.

How we test

Manual QA mostly :(with some automated acceptance tests

# Engines	Details	Team Size
18	in-repo routable Lazily	5

The good

Nothing special, they're doing their job well.

I use them for:
- splitting app into lazy loadable bundles
- mounting one route over multiple other (base) routes

I don't use them:
- as isolation or separation mechanism
- as installable third party engines/libs (sounds good in theory but not in practice)

The bad

- not being able to share more stuff (components, helpers ...) between app & engine space
- hacking the resolver to achieve better context sharing
- testing not supported well out of the box

How we test

The same as app space modules.
Dir structure is: tests/{engine-name}/unit|integration|
Had to hack the resolver to achieve that. I prefix module name with engine name which tells the resolver where to look for the module. Something like this: moduleForIntegration('company-engine/company/billing-info-form', ...)

# Engines	Details	Team Size
3	standalone routable Lazily	10

The good

Separate tests and apps and repos.

The bad

Dependency management.

Ember-data (models, adapters...) we're storing them in a shared addon. which means we have to modify there and update the dependencies.

Tests in the consuming app for the engine seem a little hard to do.

chaining addons addon => engine => app. some times this doesn't seem to work well and requires we move addons from devDependencies to dependencies. I don't understand why and when to do this.

How we test

Integrations in the engine dummy app.
visit the routes from the consuming app. only check they exist.

# Engines	Details	Team Size
5	in-repo routable Lazily	30

The good

Ability to split up the bundles for some of our larger applications. Requiring the user to wait a substantial amount of time to load the entire application bundle up front can now be avoided.

The bad

Mostly testing. All of our engine tests are located in the same directory as tests for the parent application. We have to modify the tests/index.html file to eagerly load every engine's assets up front so tests work correctly. It would be nice to store engine tests directly in the engine directory and have them be called automatically after the parent app's test when executing "ember test" in the main repo directory.

It would also be nice to have some kind of "engine loading" state provided out of the box. If an engine's assets are large, when clicking a nav item that references a route in an engine, our app currently does not show any kind of immediate feedback after the "click" while the engine assets are loading in the background. Similar to how we have loading states in routes, it would be nice to have a loading state somewhere that we can easily tap into to show a loading spinner until an engine's assets are loading (this might already exist, we just haven't investigated).

How we test

Mentioned above.

# Engines	Details	Team Size
1	in-repo routable Lazily	3

The good

Ability to improve performance for site

The bad

Addon support is not that great yet

How we test

Mocha

# Engines	Details	Team Size
1	standalone routable Eager	10

The good

It allows us to deploy our engine (sub) application alongside the host application at a much faster rate than the host application release cycle.

The bad

It would be great if there was auto discovery of routes in an engine. This is a big pain point for us as we have had to write some code generation tools to rebuild the app.js and router.js of our host application, and rebuild it if we change / add/delete routes.

How we test

Currently, manually.

# Engines	Details	Team Size
80	in-repo routable Lazily	70

The good

- Isolated code and architecture.
- Lazy load functionalities.

The bad

- Most addons doesnt support it.
- Very bad "ember-cli" design.
- Hard to extend.
- Poor dependency design.

How we test

From the app. Mostly using E2E testing outside Ember ecosystem.

# Engines	Details	Team Size
	standalone routable most eager, some lazily	1

The good

It is a long awaiting feature

The bad

Does not work with rngines route protected with ember-simple-auth https://github.com/ember-engines/ember-engines/issues/485#issuecomment-378577891

Issue reported half of year ago incuding test application to reproduce it. Still have no reply or some reaction to it. So such crytical issue ignorance also a pain preventing to use ember-engines in production.

How we test

not testing it yet

# Engines	Details	Team Size
5	both routable Some eager, most lazily	130

The good

Lazy loading!

The bad

As of now none. But there are some features which seems to be missing. (Since ember loads engine based on config, i can't find an option to add SRI in script and style urls)

How we test

no test cases for now

# Engines	Details	Team Size
78	in-repo both Some eager, most lazily	80

The good

1. Split our app in smaller parts.
2. Load these parts when the user needs it.
3. Separate each functional area in teams that works in their corresponding engines.

The bad

- Testing.
- Poor documentation for in-repo engines.
- The experimental status of Ember Engines.

How we test

We've a custom resolver (ember-cli-awesome-resolver) that allows to find anything anywhere, and with some tricks we can test all the parts (app and in-repo engines) with a single "ember test" command.

# Engines	Details	Team Size
1	standalone both Lazily	20

The good

lazyloading,
a single engine can be consumed by difference ember apps

The bad

deployment & assets compilation needs a lot customization/workarounds

How we test

with dummy app, like an addon

# Engines	Details	Team Size
12	in-repo both Lazily	5

The good

* Great separation of business logic
* Lazy-loading
* Reduced build time and size
* Helped us think about our app in a more modular and reusable way

The bad

* Sharing components between engines and main app
* Testing at first but it became easier with time
* Documentation
* Loading newly created files (might be fixed now but at the time, when we created new components or file in the engines, we had to restart server for them to be usable)
** Also, the app did not failed when using a missing component, it rendered an empty div tag which was confusing at first

How we test

Sinon, Chai and Mocha

# Engines	Details	Team Size
4	in-repo routable most eager, some lazily	4

The good

I like the theory of making it easy to break up a large app into isolated modules that can operate independently of one another.

The bad

A lot. Testing is difficult for a number of reasons I won't get into now, but the resolver makes everything more painful. Standalone engines were so painful to work with from a dev workflow perspective that we moved everything in-repo. This means we lose one of the primary benefits of engines for us which was having a separate release cadence per-engine. In-repo engines are also relegated to the lib dir which is a pain to switch back and forth between that and the app dir. All engines have to eventually use a single version of a dependency which is managed by the app's package. Code editors can't follow file paths due to Ember's resolver so many features won't work with engines (such as debugging in VS Code). The performance advantage of lazy loading seems minimal. On and on. We're in the process of removing engines and resorting to a monolith app in the hopes that tree-shaking in the build tools becomes available. If this initiative hadn't sat stagnant for so long, maybe we would've stuck with them.

How we test

We were adding a test-support directory and then running the tests as part of the main app's test suite. That gets confusing though due to how the resolver handles paths, so we've moved most of the tests directly to the app's test folder, which of course undermines the point of having engines.

# Engines	Details	Team Size
	in-repo routable Lazily	6

The good

Lazy loading and a form of code separation.

The bad

Unnecessary and duplicated configuration of services and dependencies.

How we test

Like a separate app.

# Engines	Details	Team Size
1	in-repo routable Lazily	6

The good

Only (easy) way for now to get lazy loading

The bad

Bit rough to set up in combination with FastBoot

How we test

Eagerly loads in tests, is tested via host app

# Engines	Details	Team Size
5	standalone routable Lazily	5

The good

That they allow lazy loading, other than that it's just pain in the ass!

The bad

all the setting up and pluging in based on strings and not compiled configuration is very error prone. To get it right, it's a heroic effort.

How we test

They have their own tests in their own repository. But it would be better to have those tests in main repo cuz we use them just only for lazy loading.

# Engines	Details	Team Size
1	standalone routable Eager	4

The good

Re-using of buisiness processes (bigger than individual components) across different domains

The bad

Sub-standard documentation.

Using a general component that's supposed to trigger a transition out of the engine and suddenly has to be engine-aware.

How we test

Mounted in dummy app, default ember tests.

# Engines	Details	Team Size
10	standalone routable Eager	12

The good

The bigger the application, the greater the risk of conflicts. With ember-engines the application can be split in several modules and ensure that you can maintain them without breaking others.

More over, with the use of engine, we can propose multiple application with the same base but with a different set of engines. This is a great feature when working with a product proposing multiple features.

The bad

Overrides.

How we test

QUnit test on the dummy application of the engine with ember-cli-mirage addon for mocked data.

# Engines	Details	Team Size
1	in-repo routable Lazily	10

The good

Completely abstracted part of an ember application in a reusable way

The bad

Shared services / components, *TESTING*

How we test

Acceptance tests in main application

# Engines	Details	Team Size
10	in-repo routable Lazily	10

The good

We began exploring engines as a way to combine several apps that all share the same models and some components. Engines allows us to do this, while keeping clear boundaries between the apps. We later began to use it for code splitting within apps to keep bundle sizes down.

The bad

Since engines is not "officially" supported, not all addons work (or work well) with engines. We have had a to patch several addons in order to use them with engines.

How we test

We are currently using Ember non-rendering, rendering, and application tests as well as Selenium and manual testing by our QA team.

# Engines	Details	Team Size
4	in-repo routable most eager, some lazily	5

The good

Lazy loading and code splitting by routes.

The bad

In-repo-engine tests are generated within the host app's tests folder, and not the engine's own tests folder. It would be easier to separate the engine into its own repo if the tests were also generated in the engine's folder.

Managing dependencies is also a pain point because it isn't clear if a dependency needs to be under "devDependencies" or "dependencies".

We also want to use lazy-routless engines, but that isn't supported at this time.

How we test

Unit/integration/acceptance tests within the host application.

# Engines	Details	Team Size
2	both both equal parts both	30

The good

Lazy loading... Smaller initial payload.

The bad

- Importing from node_modules was broken. Had to use a fork.
- Testing using the old syntax wasn't documented anywhere... Ran into some weird edge cases.

How we test

In-repo engine has is tested with main app. Standalone engine has it's own test suite.

# Engines	Details	Team Size
10	standalone both Lazily	50

The good

Its structured way , boundary splitting, ability to load as a separate addon

The bad

Will be better if they support components sharing

How we test

We test it manually yet to write ember test cases for engines

# Engines	Details	Team Size
1	standalone routable Lazily	5

The good

Reusable chunk of functionality, it is being used in both Admin and Client facing applications and it's keeping us DRYer than components alone.

The bad

understanding, initial setup, deployment, worries about stability

How we test

independently in the dummy application / mirage which is also deployed internally for QA..