Complete rewrite of ESLint #16557

Unanswered **nzakas** asked this question in **Ideas**

nzakas 25 days ago Maintainer edited

Introduction

ESLint was first released in 2013, meaning it will be ten years old next year. During that time, the way people write JavaScript has changed dramatically and we have been using the incremental approach to updating ESLint. This has served us well, as we've been able to keep up with changes fairly quickly while building off the same basic core as in 2013. However, I don't believe continually to make incremental changes will get us to where ESLint needs to go if it wants to be around in another ten years.

Even though we are close to rolling out the new config system, which is the first significant rearchitecture we've done, that effort is what led me to believe that it's time for a larger rewrite. We are still stuck on implementing things like async parsers and rules because it's difficult to plot a path forward that doesn't cause a lot of pain for a lot of users. This seems like the right time to stop and take stock of where we are and where we want to go.

Goals

I've been thinking about where I'd like ESLint to go next and have come up with several goals. These are pretty abstract at the moment, but here they are, in no particular order:

1. **Completely new codebase.** Starting with a completely new repo will allow us to continue to maintain the current version of ESLint as long as necessary while ensuring we are making non-breaking changes on a new version.
2. **ESM with type checking.** I don't want to rewrite in TypeScript, because I believe the core of ESLint should be vanilla JS, but I do think rewriting from scratch allows us to write in ESM and also use `tsc` with JSDoc comments to type check the project. This includes publishing type definitions in the packages.
3. **Runtime agnostic.** ESLint should be able to run in any runtime, whether Node.js, Deno, the browser, or other. I'd like to focus on creating a core package (`@eslint/core`) that is runtime agnostic and then runtime specific packages (`@eslint/node` , `@eslint/browser` , etc.) that have any additional functionality needed for any given runtime. Yes, that means an officially supported browser version!
4. **Language agnostic.** There's nothing about the core of ESLint that needs to be JavaScript specific. Calculating configurations, implementing rules, etc., are all pretty generic, so I'd like to pull the JavaScript-specific functionality out of the core and make it a plugin. Maybe `@eslint/js` ? I envision a language implementation being distributed in a plugin that users can then assign to specific file patterns. (This would replace the `parserForESLint()` hack.) So ESLint could be used to lint any file format so long as someone as implemented an ESLint language API for it.
5. **New public APIs.** Our public API right now is a pretty messy thanks to the incremental approach we've taken over the years. ESLint was never envisioned to have a public API beyond the `Linter` class (which started out as a `linter` object) and we've continued hacking on this. Right now we have both an `ESLint` class and a `Linter` class, which is confusing and they both do a lot more than just lint. I'd like to completely rethink the public API and provide both high-level APIs suitable for building things like StandardJS and the VSCode plugin and low-level APIs that adhere to the single-responsibility principal to make it possible to do more creative mixing and matching.
6. **Rust-based replacements.** Once we have a more well-defined API, we may be able to swap out pieces into Rust-based alternatives for performance. This could look like creating NAPI modules written in Rust for Node.js, writing in Rust and compiling to WebAssembly, creating a standalone ESLint executable written in Rust that calls into the JavaScript portions, or other approaches.
7. **Async all the way down.** Async parsing, rules...everything! We've had trouble making incremental progress with this, but building from scratch we can just make it work the way we want.
8. **Pluggable source code formatting.** Stylistic rules are a pain, so I'd like to include source code formatting as a separate feature. And because it's ESLint, this feature should be pluggable, so you can even just plug-in Prettier to fulfill that role if you want.

9. **Reporters for output.** The current formatters paradigm is limited: we can only have one at a time, we can't stream results as they complete, etc. I'd like to switch to a reporters model similar to what Mocha and Jest have.
10. **AST mutations for autofixing.** This is something we've wanted for a long time. I see it as being in addition to the current text editing autofixes and not a direct replacement.

Maybes

These are some ideas that aren't fully hatched in my mind and I'm not sure how we might go about implementing them or even if they are good ideas, but they are worth exploring.

- **Make ESLint type-aware.** This seems to be something we keep banging our heads against -- we just don't have any way of knowing what type of value a variable contains. If we knew that, we'd be able to catch a lot more errors. Maybe we could find a way to consume TypeScript data for this?
- **Make ESLint project-aware.** More and more we are seeing people wanting to have some insights into the surrounding project and not just individual files. `typescript-eslint` and `eslint-plugin-import` both work on more than one file to get a complete picture of the project. Figuring out how this might work in the core seems worthwhile to explore.
- **Standalone ESLint executable.** With Rust's ability to call into JavaScript, it might be worth exploring whether or not we could create a standalone ESLint executable that can be distributed without the need to install a separate runtime. Deno also has the ability to compile JavaScript into a standalone executable, so Rust isn't even required to try this.

Approach

For whatever we decide, the overall approach would be to start small and not try to have 100% compatibility with the current ESLint right off the bat. I think we'd added a lot of features that maybe aren't used as much, and so we would focus on getting the core experience right before adding, for example, every existing command line option.

Next steps

This obviously isn't a complete proposal. There would need to be a (massive) RFC taking into account all of the goals and ideas people have for the next generation of ESLint. My intent here is just to start the conversation rolling, solicit feedback from the team and community about what they'd like to see, and then figure out how to move forward from there.

This list is by no means exhaustive. This is the place to add all of your crazy wishlist items for ESLint's future because doing a complete rewrite means taking into account things we can't even consider now.

92 👍 228 😕 2 ❤️ 83 🚀 80 👀 57

25 suggested answers · 59 replies

Oldest | Newest | Top

ljharb 25 days ago Sponsor

What about AST-based autofixing? That's the sole advantage prettier seems to have over eslint.

12 👍 32

0 replies

nzakas 25 days ago Maintainer Author

@**ljharb** Oh yes, forgot to include that.

4 ❤️ 15

0 replies

brettz9 23 days ago

Very exciting....

One big picture kind of feature I'd like to see as possible would be richer AST traversal too to ensure not only `parent`, but `parentProperty`, `nextSibling`, etc. on each supplied node, as well as a scoped `querySelector` (ala esquery).

↑ 1 ❤️ 1

1 reply

nzakas 2 days ago Maintainer Author

I don't want to muddy up the AST with more nonstandard properties (I'd love to actually get rid of `parent`), but that doesn't mean we can't think of other ways to traverse the AST.

willster277 22 days ago

> 2. ESM with type checking. I don't want to rewrite in TypeScript

Valid preference, but would there be any chance of providing types within ESLint rather than in the separate `@types` package?
Currently I'm scratching my head over the flat configs as there's no type definition for them, however the options use existing type declarations. I'm trying to figure out whether to declare my own interface or to modify an existing one.

It could be useful for the large number of TS users if features had their TS declarations added at the same time they are released.
The `@types` package is at v8.2, 24 minor versions behind, which is generally a pain for TS users as things are hidden / throw compiler errors despite actually existing.

Additionally, given the desire to handle types and be type checked (all via typescript) it seems odd to not do the final little bit and simply translate JSDoc typing into TypeScript typing. In my experience I've found TS to be a lot leaner than JSDoc for declaring types, and it's also now the driver behind JSDoc types and inference.
The desire is to stay away from typescript, but also fully integrate with typescript. I understand the difference, but given the desire to support typescript features inherently, why not also support "here's a typedef so you don't have to flick back and fourth from editor to docsite"?

↑ 13 👍 41

5 replies

mrmckeb 3 days ago

Given the confidence that TypeScript provides (which is greater than JSDoc typing), another benefit of a TypeScript rewrite is that it makes it easier for people to contribute to ESLint. I always find it much easier to contribute to well typed and tested open source projects.

I'd also suggest that ESLint could go one step further and integrate TypeScript support natively into ESLint. This would make configuration much simpler for beginners, and would allow for all rules to be type-aware (optional logic within rules that add additional value if TypeScript is available).

👍 30

karlhorky 2 days ago

Yeah definitely, TypeScript syntax is way nicer than JS + JSDoc (and JSDoc doesn't support some things, so you need to use TS for these things anyway). I say this after having tried implementing a medium-size project in JS + JSDoc.

👍 12

nzakas 2 days ago Maintainer Author

I've actually found TypeScript can make it more difficult for people to contribute -- it's more cognitive overhead than plain JavaScript.

In any event, this is one area that isn't up for debate. We need to stick with plain JS so we can dogfood our core rules and processor. We'll leave it to the typescript-eslint folks to worry about TypeScript-specific functionality.

👍 17 👎 63

nikolalukovic 3 hours ago edited

> I've actually found TypeScript can make it more difficult for people to contribute -- it's more cognitive overhead than plain JavaScript.

This is an incredibly unsound statement based only on anecdotal evidence and personal beliefs. Really sad you're not taking a pragmatic approach to this issue instead of keeping to your own flawed opinions. Especially considering a rewrite of this tool is going to be a colossal work that will impact *tens of millions* of people.

👍 4 👎 1 😕 1

ivancuric 2 hours ago edited

I agree that writing in TS is less overhead than writing JSdocs.
Additionally if you drop enums, TS is entirely type-strippable with a simple transform, leaving you with pure JS.

> In any event, this is one area that isn't up for debate. We need to stick with plain JS so we can dogfood our core rules and processor.

Not sure why you can't dogfood the type-stripped version. I'm not sure that this is a good decision as I've yet to see any library that recently released that didn't eventually end up doing a TS rewrite.

ljharb 22 days ago Sponsor

Providing types in a package is a mistake; it conflates semver of the types with semver of the actual API.

↑ 2 👍 9 👎 58

7 replies

Show 2 previous replies

mrmckeb 3 days ago edited

Thanks **@stephenwade**, I think you're right - I should have had coffee before replying - I've modified my post to clarify what I was in favour of.

nzakas 2 days ago Maintainer Author

I'm not sure I understand the argument against including types in the same package as the application. Certainly, we wouldn't want to introduce type changes without some reason, and that reason likely has to do with changes in the application. Wouldn't we want the types to always be in sync with the currently installed application?

👍 13

ScottAwesome yesterday

> Providing types in a package is a mistake; it conflates semver of the types with semver of the actual API.

Is there any way you can elaborate on this being an issue?

In practice, haven't seen any real issue here. When the types update, it uncommon for it not to be a signal of API changes as well.

ljharb yesterday Sponsor edited

@nzakas there are many reasons keeping types in sync is a moving target - especially that every TS minor release is likely to contain breaking changes. Also, the TS team keeps DT types up to date *for you* (syntax-wise, not API-wise, ofc), so that's an added bonus.

nzakas 3 hours ago Maintainer Author

So are you talking about TS syntax changes?

I generate types at build times for a lot of my personal projects, so I'm just not clear on what the dangers are that you're referring too.

SrBrahma 16 days ago

I would love love love if ESLint is written in Rust. In larger codes, it's noticeable a delay on save and on Intellisense. This is definitively the future. While I love JS/TS and work with it, there is really no performance comparison with Rust. It's more of a matter of time until we have a ready JS/TS linter in Rust, not **if** it's going to happen. There was https://github.com/rslint/rslint but it looks dead already.

Also, I believe it could have a better structure to support formatters to overcome this situation: https://typescript-eslint.io/docs/linting/troubleshooting/formatting. So instead of adding prettier or dprint, we could finally have one proper tool for both linting and formatting.

↑ 20 👍 6

5 replies

onigoetz 4 days ago

There is also ongoing work by Rome (https://rome.tools) to implement a linter and some work has been done on a very basic linter in SWC (swc-project/swc#3074)

👍 5

Shinigami92 4 days ago edited

Somehow I also feel like Rust could be the better option in the long run. It's just really early days but I already see that many tools are moving to Rust for performance reasons.

Beside that: ESLint, Rome and Prettier always had one big issue in my opinion => They try to be JS/TS first.
At work we have e.g. projects using https://github.com/HubSpot/prettier-maven-plugin
And yes, under the hood it uses prettier https://github.com/jhipster/prettier-java/blob/2e0e0da2a288068c91d8f4c6133eedc4b7ce23ca/packages/prettier-plugin-java/package.json#L14
But Prettier ships with many stuff for JS out of the box which is just not used at the end, as there is no single line of JS code in our project(s).
And I highly assume ESLint and Rome is like currently the same.

I would love to use one single tool to lint everything but this tool only use what it needs and also is freaking fast. Not `20ms` per file but `20µs` per file would be a huge benefit 🚀

Another point of moving to Rust would be that slowly (at first) the whole parser / AST generators can be written in Rust which effects the whole ecosystem in terms of speed in the long run.

chaffeqa 3 days ago

I had a thought while reading this thread and wanted to introduce an option:

This may be one of those situations where there is an opportunity to unite a community by saying *the next ESLint version will be Rome* (in a sense). Basically: **join Rome**.

Rome may not be perfect, or may have some concepts you would not agree with, but I feel that could be overlooked in preference of the "greater good". I believe if ESLint puts its backing behind Rome, it may start a domino effect of movement.

What does that get the community?

- Easier decision making when starting a project
- Easier decision making on API's / config
- Easier movement for community (lets say: JS incorporates typescript as first class citizen. Single project community can rally around and make sure migrates to that new language feature)
- Combining of talents: Rome is already written in rust, and would mean less barrier to considering that option

Mostly the unification of the community is why I feel this may be a good idea. Intirested in your thoughts!

👍 5 👎 1

Shinigami92 3 days ago

I often thought about this myself, but I have my issues with Rome and their maintainer(s).
I already asked multiple times (on bird-site) if they could guide me the way of how to provide e.g. a pug plugin, but they totally ignored me.
I also looked into their docs and it feels like they want to rule everything. Also a goal of Rome is to format, lint and even bundle code but mainly JS/TS in the first place.
So also this doesn't match with my dream-linter tool.
I just want one linting tool that can lint everything on demand and only what I tell it to lint and with my configuration. Not formatting and not bundling. I already have other tools for formatting and bundling.

👍 1

nzakas 2 days ago Maintainer Author

Thanks for the insights. I'm very well aware of Rome and I don't see ESLint and Rome being a good fit to work together. Rome is all-Rust, which gives it speed but also limits the extensibility. We want to continue to support the existing (large) ESLint ecosystem, which means some parts will need to still be written in JavaScript. Plus, I don't think we want to rely on a profit-seeking startup vs. having a community-driven project like ESLint. What happens if the startup fails?

On the Rust side, I think we will definitely look at rewriting parts of ESLint in Rust. We have some folks looking at that right now. Will we go all the way with Rust? I don't think so. I just don't think we can do that without instantly making all the custom rules and plugins people have made obsolete, and then it's like starting up any other random project where people will need to get things set up again.

Never say never, but that's my current thinking. There will be RFCs when we have some more definitive ideas.

👍 6 ❤️ 9

nzakas 15 days ago Maintainer Author

@willster277

> Valid preference, but would there be any chance of providing types within ESLint rather than in the separate **@types** package? Currently I'm scratching my head over the flat configs as there's no type definition for them, however the options use existing type declarations. I'm trying to figure out whether to declare my own interface or to modify an existing one.

Yes, that was my intent. I'll update the original text to indicate that.

↑ 5 👍 2

0 replies

nzakas 15 days ago Maintainer Author

@JoshuaKGoldberg **@bradzacher** If you have time, I'd love to hear what we could change about ESLint to make working with TypeScript easier.

↑ 7 ❤️ 8

0 replies

willster277 15 days ago

> Providing types in a package is a mistake; it conflates semver of the types with semver of the actual API.

@ljharb I would say the types and the API are the same thing, if you take statically typed languages, it's usually impossible to separate the concepts simply because of the nature of static typing.
The API as you call it is less the interface and more the hidden magic, while the types are the interface; what to put in, what to expect out. Types *are* API.

Even if no types have changed, and only internal code is different, it's a good idea to maintain parity between the version tacked to the types and the version tacked to the API. "types v1.2.1" to me implies that it doesn't include any new types which may be necessary for "API ^v1.2.2".

In JS we can spilt types and API into utterly separate objects, I believe it's a bad idea to do that.

↑ 12 👍 2

0 replies

ljharb 15 days ago Sponsor

@**willster277** bumping the major version because you renamed a type is correct, but is a *very* high cost to impose on normal JavaScript consumers who have to read the changelog to discover that they're not actually affected at all.

↑ 2 😕 1

7 replies

Show 2 previous replies

bradzacher 4 days ago

Anecdotally I personally don't recall seeing a non-type-only package major bump purely for type changes unless they're major bumps.
It's rare that you are able to make breaking changes to your type API surface without also making breaking changes to your runtime API surface as they're usually pretty tightly coupled.

👍 3 ❤️ 1

ljharb 3 days ago Sponsor

I've seen it frequently - just renaming an exported argument type, for example - which has no direct JS representation - would cause this.

nickmccurdy 2 days ago

I still feel that's more of an issue with how the specific package is maintained, rather than a disadvantage to bundling types. As mentioned elsewhere in this thread, there are already statically typed languages that have in depth support for packages and updates.

ScottAwesome yesterday

> @**willster277** bumping the major version because you renamed a type is correct, but is a *very* high cost to impose on normal JavaScript consumers who have to read the changelog to discover that they're not actually affected at all.

This is given without much elaboration, can you elaborate on how this is a high cost? Also, is it actually a requirement to bump major due to types?

I just haven't seen this as an issue in practice, even at large companies (bigger than Airbnb even!) I've worked at down to smaller ones.

👍 1

willster277 8 hours ago

@**ScottAwesome** I agree. I've never found myself in a situation where I need to rename a type and make a breaking change.
Even if I had, I'd put it on the backlog until there were enough other "we really want to break this" things to warrant a breaking change.

The blindingly obvious thing to do is to do the exact same thing we do when we break our Javacript.

Before:

```
type OldNameIDontLike = /* ... */;
```

After:

```ts
/**
 * @deprecated use {@link NewName} instead
 */
 type OldNameIDontLike = NewName;

 /**
 * @since vX.Y.Z - replaces `OldNameIDontLike`
 */
type NewName = /* ... */;
```

If the content of the type has a breaking change, then simply keep the old definition under `@deprecated` and make the breaking change to the content within `NewName`.

Changes with Types work the exact same way and pose the exact same issues and have the exact same mitigations as changes with functional JS, with the key difference being the changes with Types are almost always a side effect of a change with functional JS, meaning **you would be doing this regardless of your use of TypeScript**.

👍 3

bradzacher 15 days ago edited

HOOOOO BOY. There's a lot to talk about here.

I've got a version of this written up already (typescript-eslint/typescript-eslint#5845 (comment)) but I've copied it here so that I can add more context

It's worth noting that a lot of the problems we run into with type-aware linting also apply in some degree to `eslint-plugin-import` which does its own out-of-band parsing and caching.

ESLint is currently designed to be a stateless, single-file linter. It and the ecosystem of "API consumers" (tools that build on top of their API - IDEs, CLI tools, etc) assume this to be true and optimise based on the assumption. For most parsers (`@babel/eslint-parser`, `vue-eslint-parser`, etc) this holds true - they parse a file and forget about it, and for our parser (`@typescript-eslint/parser`) in non-type aware mode this also holds true. However when instructed to use type information, our parser now breaks both assumptions - it now stores stateful, cross-file information.

Type-aware linting, unfortunately, doesn't fit too well into the ESLint model as it's currently designed - so we've had to implement a number of workarounds to make it fit - we've fit a square peg into a round hole by cutting the edges of the hole. This, as you can imagine, means there are a number of edge-cases where things can get funky.

ESLint Usecases

ESLint is used by end users in one of three ways:

1. "One and done" lint runs - primarily done by using `eslint folder` or similar on your CLI. In this style of run each file is parsed and linted exactly once.
2. "One and done, with fixers" lint runs - primarily done using `eslint folder --fix`. In this style of run most files are parsed and linted exactly once, except those that have fixable lint errors that are parsed and linted up to 11 times.
3. "Continuous" runs - primarily done via IDEs. In this style of run each file can be parsed and linted `0..n` times.

For a stateless, single-file system - all 3 cases can be treated the same! In that style of system when linting File A you don't ever care if File B changes because the contents of File B have zero impact on the lint results for File A.
However for a stateful, cross-file system each case needs its own, unique handling. For performance reasons we cache the "TypeScript Program" (`ts.Program`) once we've created it for a specific tsconfig because it's ***super*** expensive to create - so we are storing a cache that needs to correctly react to the state of the project.

Caching

These are the caching strategies that we can use for each usecase. Note that each usecase affords a different caching strategy!

1. "One and done" runs have a fixed cache - we can assume that file contents are constant and thus that the type information is constant throughout the run.
2. "One and done, with fixers" runs **mostly** have a fixed cache, except for those files that get fixed, but as fixers "***should*** not break the build", we assume that the fixed file contents won't change the types of other files.
    - This is a slightly unsafe assumption, but the alternative is to treat this case exactly the same as the "continuous" case, which means we hugely impact performance.
    - This assumption allows us to re-check a subset of the project (just the fixed file and its dependencies) with the slower builder API, rather than switching the entire run to the slower builder API - which obviously allows us to remain fast.
3. "Continuous" runs are the wild wild west. The cache has to be truly reactive as anything can change at any time and any change can impact any and all types in other files.
    - Note that by "anything can change at any time", I really do mean ***anything***. Files and folders can be created, deleted, moved, renamed, changed at the whim of the user, and most of those changes occur outside of the lint run (mentioned in more detail below)

APIs

TypeScript

TypeScript's consumer API is built around the concept of "Programs". A program is esentially a set of files, their ASTs, and their types. For us a program is derived from a tsconfig (eg the user tells us the configs and we ask TS to create a program from the config).

A Program is designed to be immutable - there's no direct way to update it.
To perform updates to a Program, TS exposes another API called a "Builder Program" which allows you to inform TS of changes to files so that it can internally make the appropriate updates to the Program.
The builder Program API is ***much slower for all operations*** than the immutable Program API - so where possible we want to use the immutable API for performance reasons and only rely on the builder API when ***absolutely*** required.

So to line it up with the aforementioned usecases - we want use the immutable API for (1) and most of (2), and fall back to the builder API when a file is fixed in (2), then (3) always uses the builder API.

ESLint's API

ESLint implements one unified API for a consumer to perform a lint run on `1..n` files - the `ESLint` class.

There are no flags or config options that control how this class must be used by consumers. This means that ESLint cannot distinguish between the above usecases. This makes sense from ESLint's POV - why would it care when it's a stateless and single-file system; all the cases are the same to them!

This poses a problem for us though because if ESLint can't distinguish the cases, then we can't distinguish the cases and so we're left with the complex problem of "how can we implement different cache strategies without being able to tell which strategy to use?"

Problems

Cache Strategy and Codepath Selection

As mentioned above, we want to use the immutable Program API where possible as it's so much faster. We do this automatically by inferring whether or not you've run ESLint from the CLI by inspecting the environment. It's a hack, but it does work for usecase (1). Unfortunately there's no way for us to differentiate usecase (1) from (2), so we have to have a fallback to switch to the builder Program for usecase (2) so that we can update the Program after a fix is applied.
If our detection code doesn't fire, we just assume we're in usecase (3), and use the slow but safe codepaths.

Slow lint runs often occur due to incorrect usecase detection due to the user running things in ways we didn't expect / can't detect (such as custom scripts), or due to cases we haven't handled.

Disk Watchers

Ideally we'd attach filewatchers to the disk to detect when the relevant files/folders are changed (would solve the "out-of-editor file updates" problem below).
Unfortunately there's no ***good*** way to attach a watcher without creating an "open file handle". In case you don't know - open file handles are a huge problem because NodeJS ***will not exit*** whilst there are open file handles. Simply put - if we attach watchers and don't detach them then CLI lint runs will just never exit - it'll look like the process has stalled and you have to `ctrl+c` to quit them.

There is no lifecycle API built into ESLint so we can't tell when would be a good time to clean up watchers. And because we can't tell the difference between an IDE and a CLI run, we can't make assumptions and attach watchers either.
So ultimately we just can't use watchers! Thus our only option is to rely on the information ESLint tells us - which is just going to be information about what file is currently being linted - and hope that is enough information.

Live File Updates

This is only a problem for usecase (3). When you make a change to file A in the IDE, the IDE extension schedules a new lint run with the contents from the editor which we use to update the Program. If you have file B that depends on the types from file A, this means that we've also implicitly recalculated the type for file B.
However, the extension controls lint runs - so we cannot trigger a new lint run on file B. This means that file B will show stale lint type-aware errors until the IDE schedules a new lint run on it.

Single-threaded vs Multi-threaded linting

The implicit update of file B's types based on changes to file A assume that both file A and B are linted in the same thread. If the aren't linted in the same thread, then updates to file A will not be updated in file B's thread, and thus file B will never have the correctly updated types for file A - which leads to incorrect lints!! The only way to fix this would be by restarting the IDE extension (or the IDE itself!)

Out-of-editor File Updates

In all IDEs it's possible that you can use the "file explorer" to move files around to different folders, or even rename files. This disk change happens outside of the editor window, and thus no IDE extension can or will tell ESLint that such a change occurred. This is a big problem for us because the Program state explicitly depends on the filesystem state!

We have some **very slow** fallback codepaths for this case that attempts to determine if out-of-editor changes occurred on disk, but it's not perfect code and can miss cases.

So with all that being said... what would I want to see from a rewritten version of ESLint?

Well the biggest problem we have is that we cannot tell what state ESLint is running in, so we have to rely on fuzzy and unsound logic in order to determine cache strategies.

So I'd really want ESLint to be able to tell parsers and plugins about the state ESLint is running in so that they can make decisions about how to invalidate or update their data-stores.
I suspect this means that ESLint will need to have more than one API for consumers instead of the single `ESLint` API it exposes - but that's something that can be nutted out later? Hard to say.

Worth mentioning this is something I've been thinking about for a long while (eg #13525), but obviously haven't had the time to do any formal design or RFCs.

6 replies

Show 1 previous reply

nzakas 2 days ago Maintainer Author

Okay, so I've had time to digest this and work it into my working theory of where ESLint should go in the future. Here's what I've come up with:

1. ESLint should have a notion of a *session*. A session is a single run of ESLint, encapsulating all files to be linted and whether or not files are being fixed. As an abstract idea - what if there was a `context.session` object such that you could check

`context.session.filePaths.length > 1` and `context.session.mode === "fix"` ? I think that would solve your problem of knowing how ESLint is currently being run?

2. Once we have a concept of a session, we could expose events like `SessionStart` and `SessionEnd` that a plugin could listen for to know when to trigger setup/teardown.

This is just a really back-of-the-napkin idea, but what do you think?

@**ljharb** would this also help with `eslint-plugin-import` ?

👍 3

bradzacher 2 days ago

> what if there was a context.session object

That would work - as long as we pass that to the parser as well as the lint rules!

It's the sort of thing I was thinking yeah - if you know what ESLint is running on then you can make decisions around resource management ahead of time. Eg if we know that all the files for a project have been linted we can delete a project from memory and free up space!

Knowing if we are potentially applying fixers allows us to accurately handle that de-optimisation case, which is good!

And having an event model seems like a good idea that would allow watchers (which solves a lot of problems).

👍 4

JoshuaKGoldberg 2 days ago Sponsor

Passing this to the parser is actually what we've been off-and-on talking about sending as an RFC to ESLint for some time! It's what I've been planning on working on once our v6 is out the door.

ljharb 2 days ago Sponsor

@**nzakas** absolutely that would help; as long as individual rules could register their own listeners. If the identify of `context.session` remained consistent, that could be a WeakMap key for anything that needs caching per-session :-)

👍 5

nzakas 3 hours ago Maintainer Author

@**bradzacher** @**JoshuaKGoldberg** for clarification: you're saying the session info is needed not just in rules and plugins, but also in the parser? Can you explain more about that?

@**ljharb** ah that's interesting! I hadn't thought about exposing these events within rules. Can you explain more about how you think that would work? My initial thought was that plugin-level hooks would the best use but definitely want to hear more.

JoshuaKGoldberg 13 days ago Sponsor

> Language agnostic

If this is intended to be used for any language -either specifically in the web ecosystem or more broadly-, why not rewrite in a more performant language -- e.g. Rust?

↑ 13

6 replies

Show 1 previous reply

oliviertassinari yesterday

edited

I would love to see markdown support. I would be a good complement to prettier. See the rules in https://github.com/DavidAnson/markdownlint.

MichaelDeBoey yesterday

@**oliviertassinari** You can already lint markdown using the official `eslint-plugin-markdown`
https://github.com/eslint/eslint-plugin-markdown

oliviertassinari yesterday edited

> eslint-plugin-markdown

This seems to lint JavaScript inside markdown. I want to lint markdown.

bradzacher 22 hours ago

It's possible to support any language with ESLint right now (eg `eslint-plugin-relay` or `eslint-plugin-graphql` lint graphql fragments and `.graphql` files), there are just certain constraints the parsers need to work within to produce an AST that ESLint won't crash on.

@**oliviertassinari** there are things like `markdown-eslint-parser` and `eslint-plugin-md` but as you can see from the download counts - in general people will just get behind tools that are built for purpose (like markdownlint or remarklint).

nzakas 3 hours ago Maintainer Author

To clarify: a lot of these other tools copy ESLint mechanisms to bootstrap their own linting. The idea here is to create a core that is general purpose both to make it less difficult for things like graphql-eslint to plug in and also to allow folks to more easily create standalone tools like markdownlint that currently need to reimplement all the boring stuff the ESLint core has.

👍 3

kecrily 13 days ago Collaborator

It is recommended to convert this issue to discussion. We need structured comments.

↑ 2 👍 5

1 reply

nzakas 9 days ago Maintainer Author

Done!

🚀 5

ljharb 13 days ago Sponsor

@**JoshuaKGoldberg** a major downside of writing any JS tool in "not javascript" is the dramatically decreased pool of potential contributors.

↑ 11 👎 3

8 replies

Show 3 previous replies

nickmccurdy 4 days ago

But with Rust's ability to call out to JavaScript (as mentioned in the OP), wouldn't there still be the option to do the same thing in JavaScript?

👍 3

karlhorky 2 days ago

It will still be "not JS" because of the extra JSDoc syntax that you'll be introducing (which is worse than TypeScript, having tried this out).

So better to introduce a new language that has a better syntax, more features and a larger developer community (TypeScript) than one that is worse on all of those metrics (JSDoc)

👍 3

nzakas 2 days ago Maintainer Author

@JoshuaKGoldberg

> So just confirming - although ESLint will not be JS-specific, the idea is that it still is a JS-first tool? E.g. you might be able to use it for non-JS code such as Markdown or even any arbitrary language, but the focus will still be on JS/TS & adjacent web languages?

Sort of. The idea is that ESLint will be targeted primarily at the web development ecosystem. The ESLint team will still maintain JS-specific functionality and possibly some others (JSON seems like an obvious one?). JS still seems like the best language for plugins and custom rules, so likely that the core will remain written in JS. But we really want to encourage others to create plugins for other related languages like CSS, Markdown, TOML, YAML...anything that is typically found in web development.

> I'm under the impression that if someone wanted to write a plugin in JS and have it be called by Rust, they could.

Yes, this is possible, however, you pay a penalty every time you cross over the Rust-JS boundary. If you think about how rules work, where you create an object that then visits nodes in the AST, it's not a very clear line between Rust and JS. Let's say the AST lives in Rust, that means for every visitor function we call, we'd end up serializing the AST node from Rust to pass into JS. That's some non-trivial overhead. So, that likely means the AST needs to stay in JS to avoid paying that cost. And if the AST stays in JS, there are likely other things that need to stay in JS to avoid crossing the boundary multiple times. So, short answer: this is more complex than it seems. If ESLint plugins just provided data without functionality, we would have more options, but as it stands, a complete Rust rewrite seems not to be the best approach. That said, using Rust and embedding Deno allows us to start with most of the core in JS and then to slowly try to replace pieces with Rust to see what will work and what the performance implications are.

And as **@ljharb** mentioned, a complete switch to Rust would limit contributions. Having spent the past month learning Rust, I can tell you it is very intimidating and often very frustrating. Given that we are targeting the webdev ecosystem, JavaScript is a much softer landing point for contributions.

👍 3

onigoetz 2 days ago

> Yes, this is possible, however, you pay a penalty every time you cross over the Rust-JS boundary.

> So, that likely means the AST needs to stay in JS to avoid paying that cost.

I agree with these statements but it makes me wonder, if the parts that handle the AST (by that I understand parsing and linting). What are the other pieces that could be written in Rust and would not have to pay this boundary-crossing cost ?

I've seen many projects talk about having parts using Rust, other languages, either binaries or WASM that were disappointed of the performance wins from the native parts as they were offset by the serialization/deserialization cost when crossing the boundaries.

Here is for example the discussions that happened for SWC:

- a first plugin implementation : **Javascript plugin** swc-project/swc#471 (was removed since but can't find the related PR)
- a discussion to add something new for V2 : **[Meta] Plugin system for 2.0** swc-project/swc#2635 (didn't land, went for WASM plugins instead)

On another note, Parcel made an interesting use of SharedArrayBuffers to be able to share maps across threads, since SharedArrayBuffers are also available in Rust, I wonder if that could help with serialization/deserialization issues (disclaimer: I don't have any significant experience myself with WASM or Rust)
parcel-bundler/parcel#6922

nzakas 3 hours ago Maintainer Author

It's not about avoiding the boundary cost, but minimizing it. For instance, a lot of the CLI bootstrapping can easily be done in Rust and be a lot faster than in JS: searching the file system, reading files, etc. Then we can just pass those strings into JS to work with.

There are a lot of possibilities and a lot of existing approaches to review. Right now we are just capturing ideas, so I can't be more specific than that.

JoshuaKGoldberg 4 days ago Sponsor

> Make ESLint type-aware ... TypeScript

One potential-maybe problem is linking the concept of *type awareness* to *TypeScript specifically*. I worry that the community is starting to enable typescript-eslint's APIs in shared packages so much that we're making extracting ourselves from TypeScript difficult. TypeScript has issues in its control-flow analysis that are only likely to be solved by a native-speed equivalent. We're starting to see early-stage TypeScript competitors pop up, such as Ezno and stc.

One path ESLint *could* go down is:

1. Adjust the core structure to allow for type-aware linting via a plugin (i.e. **@bradzacher**'s comments here: **Complete rewrite of ESLint** #16557 (comment))
2. Create a standardized API around type comparisons that plugins can plug into (i.e. **Proposal: Type Relationship API** microsoft/TypeScript#9879, but generalized)

↑ 6 👍 1

5 replies

nickmccurdy 4 days ago edited

Could we eliminate or simplify the problem by using the Type Annotations proposal? From what I understand, its syntax and semantics are still incomplete, but the general idea is to make most TypeScript-like syntax valid JavaScript. ESLint should be able to support a standard AST with types, then we could use discriminated unions or plugins to represent types in specific type checkers (TypeScript, Flow, Ezno, etc.).

bradzacher 4 days ago

honestly - the type annotations proposal is... weird. It's just a proposal for a sort-of-structured place in JS code where type annotations can go, but there's no spec for even basic validation (like, say, track that types reference types, or names reference things that exist), let alone type validation.
So ESLint would have to design, spec and build an entire system around that complexity.

It's also a long, long way off (if it ever actually lands).

I think that making the type-information portion pluggable is a great idea because it means ESLint is agnostic of the type-system, but provides a consistent API that future parsers could build within to allow plugins to opaquely consume type information.
Again though it would require very careful design to ensure that it's providing a truly system-agnostic API.

👍 5

nzakas 2 days ago Maintainer Author

Yes, my intent was to make type-awareness as generic as possible and not specifically tied to TypeScript. Allowing it to be pluggable would make a lot of sense.

My feeling is that there will likely be some tools developed with non-TypeScript type awareness and it would be good to be able to hook that in. I could see a situation where the core JS plugin has a default type-aware functionality (using JSDoc, most likely) that could be swapped out.

👍 2

JoshuaKGoldberg 2 days ago Sponsor

Amusingly, JSDoc types already exist in TypeScript. There might be a lot of time saved by going with TypeScript as a default provider for that system - both in developing the system, and for users needing to set it up.

👍 3

nzakas 3 hours ago Maintainer Author

Yup, that was my intent. But to your other point, figuring out a way to still make that plug gable would be important.

xavikortes 4 days ago

edited

Sounds great!! One thing I think that can be improved (not necessary related with the reimplementation) is the mess up with the config files in the root of the project. Maybe ESLint could start a new standard of .config folder or similar.

↑ 3

1 reply

nzakas 2 days ago Maintainer Author

Once the new config system is in place, we are going to stick with it. You an always specify an alternate config file location using `-c` on the command line if you want to move your config file elsewhere.

nickmccurdy 4 days ago

edited

I'd like to emphasize the desire for designing the rewrite around improved performance.

The faster feedback loop tools like ESLint offer is already valuable, but it is still fairly performance bound, especially when used in project-level rather than file-level use cases. ESLint's currently slow feedback loop means that file-level lints tend to be preferred (via editor plugins or git hooks), however overall performance could get worse if we add more project-level information to ESLint. Rust and type support could definitely improve this, but I'd like ESLint to be redesigned at its core to encourage performance patterns (like 11ty/Vite/Vitest/etc.).

Additionally, focusing on performance can potentially reduce fragmentation when competing with other tools. For example, the popular formatter Prettier is well supported but fairly slow, and Rome provides a faster Rust-based alternative to ESLint. By providing a fast and extensible platform for AST analysis, we could possibly make faster alternatives to tools like Prettier while including/maintaining extension support.

↑ 5

1 reply

nzakas 2 days ago Maintainer Author

Yes, performance is a main driver of this rewrite.

👍 3

kvz 3 days ago

As a user (encouraged to share my point of view as one), I am very happy with ESLint! ✨ If it can become faster, and if TypeS(cript) could become first class citizens, I'll take it. Other than that I really have no desires :) Thank you for making all of our code better Nicholas 🙏

PS as a QoL suggestion just for yourself, perhaps you could reconsider on making ESLint in TypeScript/Rust after all, it will be more expressive and enjoyable to write than if you'd use JavaScript/Rust and sprinkling JSDoc types on top imho. `sucrase-node` will keep your dev iterations ~~blazingly fast~~ free of heavy build steps. Those build steps you save for CI and npm releases, so that you can still ship that vanilla JS (plus types for free) to consuming devs.

↑ 7

1 reply

nzakas 2 days ago Maintainer Author

Already mentioned above, but TypeScript is off the table. ESLint needs to work on vanilla JS out of the box, and to do a good job, we need to dogfood ESLint on itself. We'll leave TypeScript-specific functionality to the typescript-eslint folks.

👍 4 👎 8

Relequestual 3 days ago

Project lead for JSON Schema here. I know you use JSON Schema implementation `ajv`. When you come to determining if you still want to use JSON Schema, and what implementation you might use, I'd invite you to come have a discussion with the JSON Schema team/community.
Newer versions of `ajv` have some issues which we won't go into here.
Newer versions of JSON Schema support far easier extensibility.
IMHO, it's worth investigating if that extensibility may make it easier for plugin authors to define the additional config options they want to have, AND provide auto-complete / intelisense / further information about specific keys and values in a config file.

I'm keeping it short so it's more likely to get read, and I realise it's probably a SMALL element of concern, but given the consideration is a rewrite, now is the time to say something and offer our assistance. =]

↑ 5

1 reply

nzakas 2 days ago Maintainer Author

Thanks, we can consider this. The big problem we have is compatibility -- if we don't want to force everyone rewrite their existing rules to use a new schema format (which we don't), we have limited options. This is also why we never upgraded `ajv` once the next major version came out -- it had too many breaking changes that would have caused a headache for the ecosystem.

karlhorky 2 days ago

edited

Amazing to hear this proposal! A lot of new great things here 🙌

However, another vote for reconsideration of rewriting in pure TypeScript (not JS + JSDoc), because of things that were already mentioned:

1. Simpler syntax: TypeScript annotations are much simpler in a lot of cases
2. Features: some things are not possible alone with JS + JSDoc (for these features, some users decide to mix in some extra TypeScript - at that point, you have 3 languages!)
3. Community: JS + JSDoc has a much smaller community and public documentation (blog posts, official docs, etc). This makes it much more challenging to find information about how to do things.
4. The "more contributors with JS" argument: you're introducing a new language anyway (JSDoc), and contributors will not be allowed to break types. So either they learn a more complicated syntax with much smaller community, or they learn TypeScript, which is better on those metrics. Also, "more contributors with JS" is becoming less of an argument over time, with TS community size making consistent gains on JS community size.

Saying this after having implemented a medium-size project in JS + JSDoc and regretting it every moment.

↑ 16 👍 13

2 replies

nzakas 2 days ago Maintainer Author

As already mentioned in several threads above, TypeScript is off the table. I'm aware of the tradeoffs but we need to dogfood ESLint in vanilla JS.

👍 4 👎 4 😕 2

arcanis 30 minutes ago

Just to throw my own two cents, I think the dogfooding argument should actually go towards using TS: while ESLint works very well for raw JS, the way it works for TS isn't as smooth. I would imagine if ESLint was written in TS (and thus was dogfooding it), the integration would be much better (at no cost for JS, I'd also imagine, with JS being essentially a subset of TS).

👍 1

bradzacher 2 days ago

edited

One other thing that I've been thinking about is the idea of parallel linting and cross file information.

For a purely single file system it doesn't hugely matter how you bucket files together across threads because each file is independent.

However for a system with cross-file information you need to be context-aware in how you bucket files - you want to keep as many related files together as possible so that you don't waste time duplicating work across threads.

Foe typescript-eslint it's super simple to do this bucketing - you do it at the project level. Each tsconfig represents a unit of code that needs to be together or else you need to duplicate the `ts.Program` in multiple threads (which is the most memory and time intensive piece of the type-aware parse!).

So having some notion of "parser-informed bucketing" would be really good for the future state of eslint.

It's worth noting that such a system would also benefit single-threaded linting as well! Why? Well right now typescript-eslint has to treat eslint's parsing as "random access", even in single-run mode. We don't know what order ESLint will ask us to parse files in AND we don't know what files are being linted AND we don't know if a file will be linted multiple times.

This is an issue because it means there's no point at which we can drop a project from memory because we don't know if we'll need it again - so we just accumulate memory as we setup more projects over time. For large workspaces this means we will eventually cause node to OOM.

If we could hint to eslint that we'd prefer if the files are linted in a certain order such that each project is linted to completion before starting on the next project. In conjunction with the above mentioned "session context", we'd be able to know exactly when the current lint run is "done" with a project. This in turn means we can ensure that we keep exactly one project in memory at a time - ensuring we're not going to cause an OOM!

Worth mentioning that I have played around with the idea of parallel linting bucketing by project and it works really well, and I saw some not insignificant perf wins from it!

This PR includes a proof-of-concept CLI for typescript-eslint which sits in front of eslint to split work across threads where each thread is essentially just `eslint <...files from project>`.
typescript-eslint/typescript-eslint#4359

↑ 5

1 reply

nzakas 3 hours ago Maintainer Author

> So having some notion of "parser-informed bucketing" would be really good for the future state of eslint.

Do you have an idea of what this might look like?

> If we could hint to eslint that we'd prefer if the files are linted in a certain order such that each project is linted to completion before starting on the next project.

There's an interesting tension here — ESLint currently works on files, not projects. If ESLint gets a list of files to lint, what would happen after that? Pass that to you for you to further dig into the file system? And what about the browser playground? Without a file system, what would change?

kdy1 22 hours ago

Hi. swc author here.
Actually, swc has `swc_ecma_lints` and has lots of rules, but as the swc core team is quite small, we decided we can't maintain it, and I'm fine with handing it off. So it may help.
Feel free to contact me (even by email or twitter/discord dm) if you are interested.

↑ 11 ❤️ 1

1 reply

nzakas 3 hours ago Maintainer Author

Thanks! I'm afraid we also don't have the bandwidth to maintain a separate project.

This comment has been minimized. Sign in to view

This comment has been minimized. Sign in to view

lloydjatkinson 26 minutes ago

Not great to "minimize" comments to hide their upvotes.

↑ 5 👍 2

0 replies

Category

💡 **Ideas**

Labels

core needs bikeshedding breaking tsc agenda needs design

28 participants

and others

Converted from issue

This discussion was converted from issue #16482 on November 16, 2022 18:54.