@harudagondi

A Rust Documentation Ecosystem Review

Reading time: 100 min read

Table of Contents

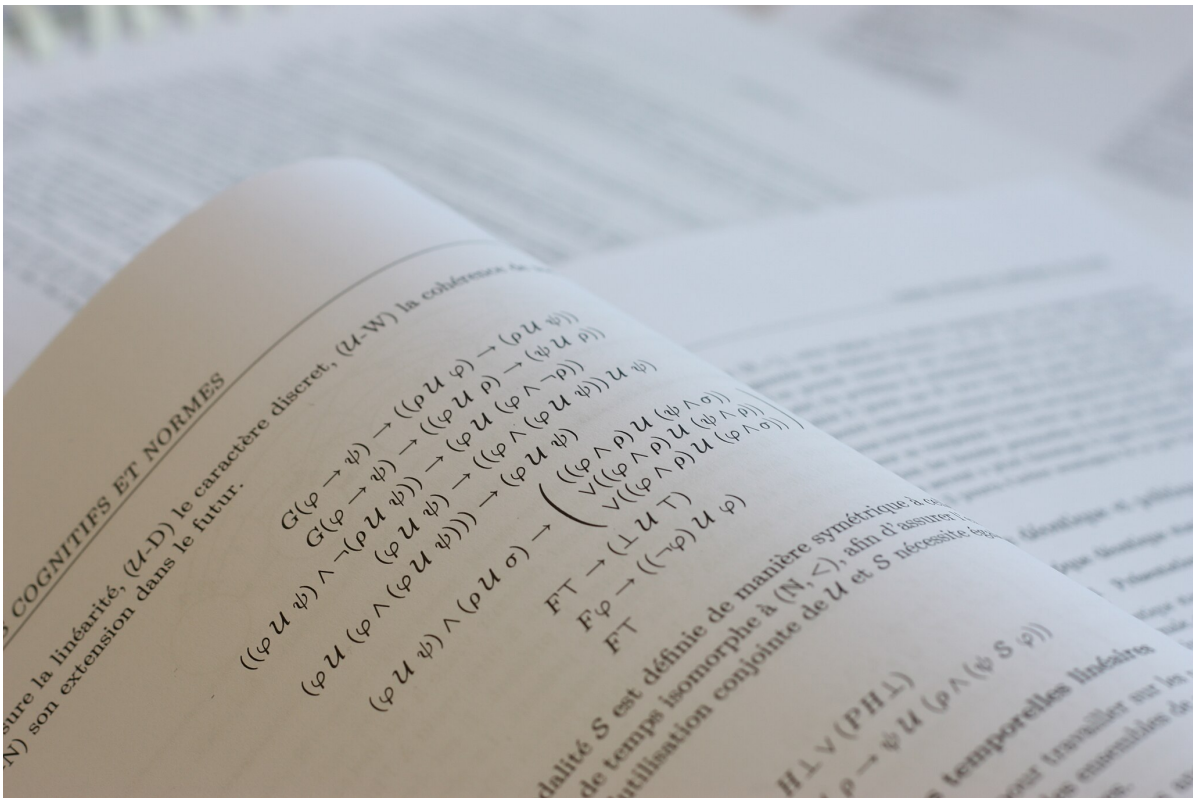


Photo by Guillaume Piolle.

I've been thinking about documentation lately.

Anecdotally, documenting code is somehow one of the least favorite things programmers do. when developing. However, it is undeniable that documentation is essential when it comes to teaching beginners. In fact, one of the most typical ways people can begin their journey in programming is by following a YouTube video tutorial, or reading a book, or tinkering with examples in the "recipe" of your choice. Without such resources (or access to

such), less people will be introduced to the field of programming and software development.

Clearly, documentation is important. We read manuals. We read books. We watch video tutorials. And yet, it is an afterthought of a lot of developers, especially those who publish libraries for people to use. *Especially* for people who want such libraries to be popular. Reading documentation is the way for beginners to learn, and the way for experts to remember.

But what constitutes good documentation?

Documentation quality is something that's not very easy to measure. There are no guidelines to determine whether or not documentation qualifies as satisfactory to the intended audience. However, there are some signs that can tell whether the documentation is *bad*. An obvious example is the *lack* of documentation. How would people understand your library if you don't have the means to teach it?

There are also other, more *subtle* ways why someone could say the documentation is bad. In newer libraries, the documentation they have is just the API reference. You could glean the meaning of each function and each class and each struct and each record, but the way it can be put together is non-obvious. How do you use so-and-so function? What's the entry point of this library? Why was this framework architectured this way?

Sometimes, when I don't see the answer in the documentation. I'll have to resort to asking people on Discord. I'll get an answer, I'll say thank you, and then I don't think about the Discord server for at least three years. But then, I would think about the next person who would ask the same question, and if they would think twice about using Discord. We have to admit that Discord isn't an accessible place. It's a walled garden, and the search functionality is dead. Forums are a much better solution to this, as it has better search indexing and more prominent, but it's rare to see it in some places.

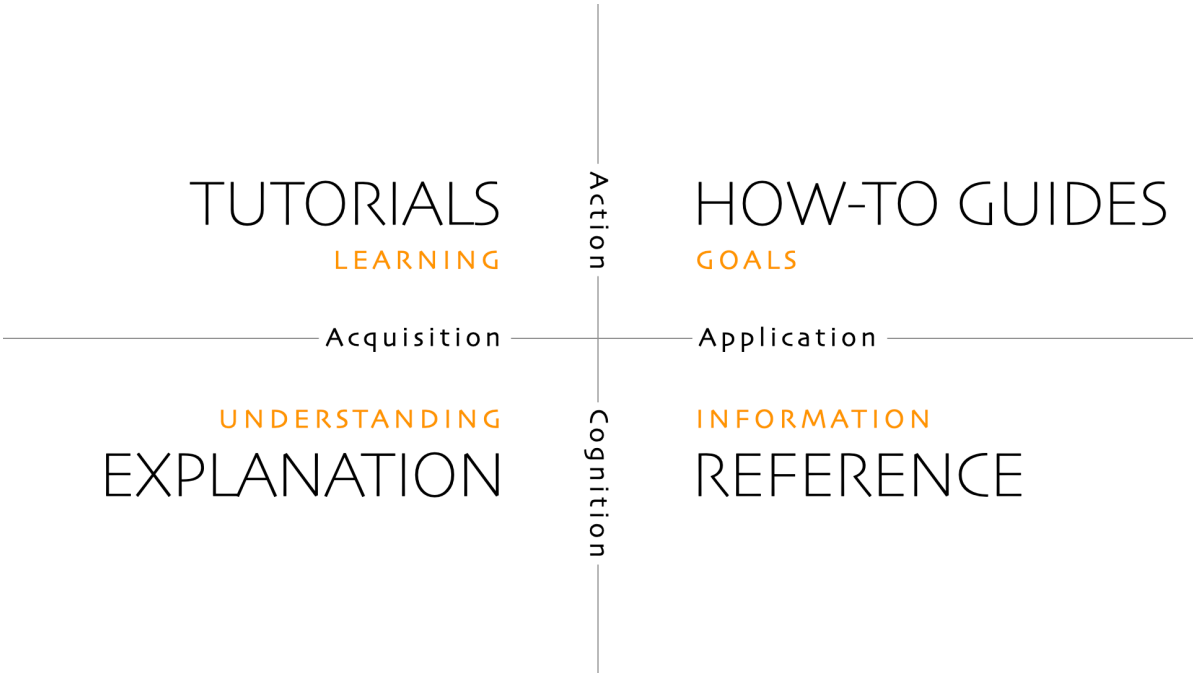
Ah, discoverability. It's hard to discover new things without either reading the whole API reference or consume several hundred hours worth of tutorials. Do I need to read the newsletter for each new added feature? Do I need to see some obscure esoteric code in some random Discord channel? There were a

lot of cases where people discover and rediscover the `@` syntax after months of dabbling in Rust and I was kinda surprised, since it was actually discussed in the Rust Book on the chapter of patterns. Is the Rust Book not approachable enough?

I'm sure there are many other criteria that determine the quality of documentation, but I want to focus on these three, based on the previous anecdotes I've stated:

1. Comprehensiveness — the amount of substance on the topic
2. Discoverability — how easily can you explore new topics
3. Philosophy — knowing the reasoning of the choices made in the library
4. Approachability — is the documentation intimidating or not

As I've been thinking of such topics, I came across this image somewhere:



It seems interesting, as it neatly categorized documentation to four topics. Let's investigate what is the meaning of this image, shall we?

What are the four quadrants?

I've tracked this image down. It's from the site called diataxis.fr. According to the website, the term *Diátaxis* refers to the "systematic approach to technical documentation authoring". It's derived from the Ancient Greek *δῐᾰτᾰ́ξῐς*, "dia" meaning across and "taxis" meaning arrangement.

It mentions that it solves the knowledge management problem by prescribing a system of four quadrants: *tutorials*, *how-to guides*, *technical reference*, and *explanations*. I recommend reading the whole website to fully understand what each quadrant entails, but to summarize:

1. A tutorial is a *practical learning experience, where the user learns by doing*.
2. A how-to guide is a *manual that addresses a real-world goal or problem*.
3. A reference is a *factual collection of documentation that contains the technical description*.
4. An explanation *describes the philosophy and provides context on why something is so-and-so*.

Additional, the *Diátaxis* compass (referring to the image above) shows the relationship between each system. Tutorials and how-to guides are driven by *action*; what the user *does* takes the driver's seat in this scenario. On the other hand, references and explanations are driven by *cognition*; what the user *knows* is the main character of this story.

Action vs. cognition is not the only axis we can extract from this graph. Tutorials and explanations are useful when a user wants to *acquire* the knowledge and skill required for the usage of a library, while how-to guides and references are for when a user wants to *apply* said knowledge and skill to the task at hand.

The website has a more comprehensive overview of what constitutes good documentation for each variant, but for now I'm not going to delve into that. Instead, I'll discuss it later as needed when we survey the Rust ecosystem's approach to documentation.

Why are they important? And is it worth it?

I want to return to the criteria of what I believe is important in documentation: comprehensiveness, discoverability, and philosophy. How do the four quadrants measure up to these criteria (that I just made up lol)?

Tutorials

In the context of the *Diátaxis* system, tutorials are not comprehensive at all. It operates on a "learn what you only need" basis, and thus it does not aim to discuss the entirety of what can be discussed about a library. As such, its main purpose is to impart only the necessary knowledge without getting into the weeds of things. This is not to say that tutorials are useless; it's just that it never aims to be comprehensive in the first place.

What it makes up for it is the discoverability. Tutorials are typically for complete beginners. It introduces the main API needed to get started and get going with the library. It allows the user to know what and where to look for. And of course, it solves the problem of "you don't know what you don't know". By knowing the main and important interfaces needed to interact with the library, it jumpstarts your journey and allows the opportunity to go deeper.

However, tutorials are not good for explaining the philosophy of the library. A tutorial serves as a learning experience, and I would say *learning* is not the same as *understanding*. You don't need to understand why a function looks a certain way, or why a codebase is architected like that. A user may just simply want to *do* something, and the method to learn how to do it is simply a linear no-frills guide that takes you from point A to point B. It is not a realistic depiction of what to do when there are unexpected outcomes. It does not diagnose. It is contrived. It doesn't give choices. It is easy to do, and therefore useful for learning the methods but not understanding the philosophy.

Lastly, by its nature, tutorials are very approachable. It assumes you know nothing. It guides your hand. It handles all possible situations for you. You just

have to follow the steps and get a product. Simple, easy, fast.

How-to guides

Unlike tutorials, which is aimed at people who doesn't have knowledge to start using the library, how-to guides is typically targeted to people with at least basic competence on the topic at hand. They are not for learning the tools, but for achieving goals with those tools. It's not for studying, it's for application in real-world scenarios. As such, how-to guides are typically more comprehensive since it discusses beyond what a tutorial would discuss. As the Diátaxis website discusses, it serves your *work*, not your *study*[†] . A real-

[†] A good guide to distinguish between the two is that tutorials discuss *the basics* and how-to guides discuss *the advanced*

world situation is more complex than idealized scenarios. As such, the documentation shall be appropriate for that situation, and discusses the possible mishaps and unexpected events that may happen, plus the appropriate tools for those.

As a natural extension, how-to guides has more discoverability when compared to tutorials, given the nature of anticipating the user and unexpected outcomes. It has to give more information depending on the scenario. This necessitates introducing new API that may not be introduced in a tutorial, for fear of overwhelming the user with complexity.

Due to the goal oriented format of how-to guides, the explanation of why an interface is the way it is is not at the forefront. As such, like tutorials, it omits details for understanding and just gets straight to giving you the answers.

Unlike tutorials however, how-to guides expect basic competence. It does not hold your hand. It doesn't idealize scenarios. It assumes familiarity with the subject. It gives you choices that you have to make. Therefore, for a beginner, it is less approachable compared to a tutorial, but for someone who knows their stuff, how-to guides may be more up their speed.

References

Ah. the most common documentation format and the least approachable. It is good for learning a library in terms of its parts, but it is not a good format for

learning a library in whole. You don't really know how to start, nor how to solve problems because a reference does not typically give you the whole picture. It is useful for examining its items, less useful in extracting the relationships between items, and least useful when developing a program out of its entirety.

It is, for what its worth, maximally comprehensive by design. It describes each item in detail (hopefully), and answers any and all potential questions by the user. Does it throw an error? If so, in which scenarios and what corresponding errors does it return? Is there any special behavior I have to take note?

However, it does not explain its philosophy very well. References are, by design, cold, and exacting. No ambiguities, totally authoritative. As the Diátaxis website states: "One hardly *reads* reference material; one *consults* it." It is not the *why*. It is the *what* of the library.

I guess its discoverability is both at its maximum and minimum. Maximum, because you can discover absolutely everything here. Minimally, because it is not a good way to naturally discover anything. There's no motivation that is presented here, and so there's no reason to discover anything yet.

Explanation

Based on my experience, the explanation is probably one of the rarest format of documentation there ever is. It's rare that someone would write about the reasoning why they coded it a certain way. You'll just see the changelog be like "Renamed `DistString` to `SampleString`" whose motivation is stated in the PR. Rare will you see an official writeup why the maintainer decided to architected it in some way, and why it was necessary to do so.

Continuing with analyzing each documentation format by my own criteria, explanations are approachable for people with basic competency and teaches its philosophy very well. Teaching the reasoning is what makes explanations so useful. It gives the user a concrete way to grok things instead acting like the library is a black box. It lessens the frustrations of beginners, as it goes beyond the common response of "it is what it is." It gives the motivation of the library, why it exist, why it works like that, and why not like this other way. It is the greatest tool for understanding things.

It can be a tool for discoverability and comprehensiveness. The author/maintainer of the library has the opportunity to showcase what the ideal program you could make with their library, and what functions and types they showcase may not be familiar to you. The maintainer can unfold the secret of the machinery and possibly introduce you to some topics that is not normally talked about in everyday usage.

To recap...

Here's a simple table on what I think each documentation quadrant can provide, based on the criteria I devised:

	Tutorial	How-to Guide	Reference	Explanation
Comprehensiveness	⚠	⚠	✅	⚠
Discoverability	✅	✅	⚠	⚠
Philosophy	❌	❌	❌	✅
Approachability	✅	⚠	❌	⚠

Legend:

- ✅: Achieves the criteria
- ⚠: Has some lapses
- ❌: Does not achieve in any good way.

At a glance, tutorials and how-to guides are very similar, differing from their approachability. However, the issues I have with in terms of comprehensiveness are different. Tutorials are not comprehensive in the sense that it does not introduce you to every idea ever, but it is comprehensive in that it has anticipated every problem the beginner might face because that would be the author's responsibility to help you. On the other hand, how-to guides are more comprehensive than tutorials because of its increased complexity, but it falls under the same situation where it doesn't necessarily introduce you to the entire library.

References are maximally comprehensive, as by design, but it is the only

quadrant that is not approachable by any means. Accordingly, It is very useful for consultation of topics but not learning of said topics.

The Rust Ecosystem Review

Now, with what we've learned about the four quadrants and criteria I have on hand, we can survey the Rust ecosystem and see if they hold up to our standards. I will pick a selection of crates from the blessed.rs website, with some additional crates not mentioned that I think could be useful for comparison. To start, here are the crates I've picked:

1. `rand`
2. `fastrand`
3. `chrono`
4. `time`
5. `jiff`
6. `axum`
7. `actix-web`
8. `rocket`
9. `bevy`
10. `macroquad`
11. `fyrox`
12. `thiserror`
13. `anyhow`
14. `snafu`
15. `clap`
16. `pico-args`
17. `ratatui`
18. `egui`
19. `iced`

20. `wgpu`
21. `tokio`
22. `smol`
23. `embassy`
24. Rust's official documentation

I'll be measuring the quality of their documentation by the criteria I've set and checking the existence of the documentation quadrants I've discussed beforehand. I will not, however, be measuring the quality of the code or the API of the library. The goal here is to check if the ecosystem matches my expectations of what good documentation should be.

`rand`

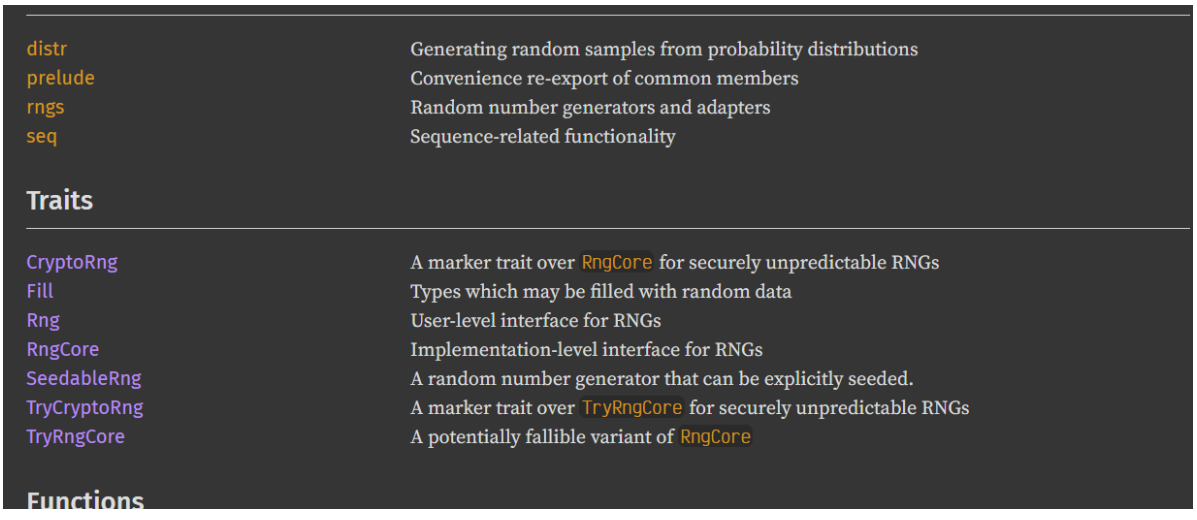
`rand`[†] was the first on the list of the blessed.rs website. So naturally let's look

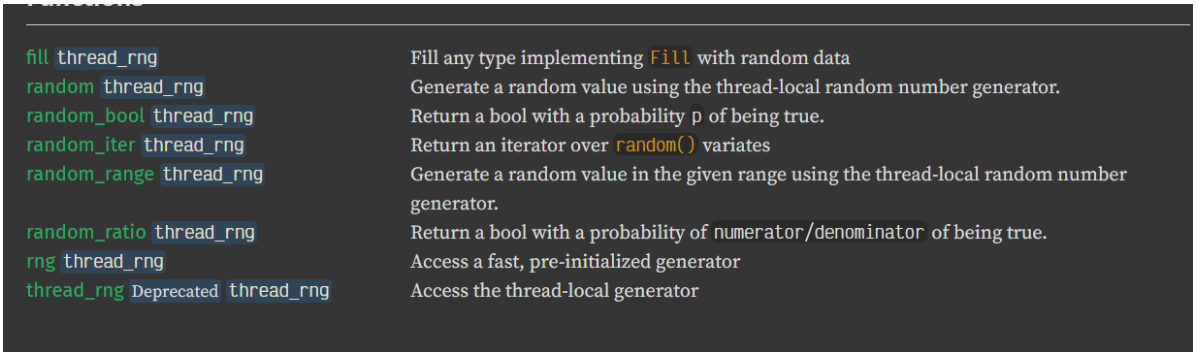
[†] As of writing, the version is 0.9.1

at it first.

The crates.io page has a general overview written, linking the different random number generators (RNGs), its features, anti-features, and platform support. Seems reasonable for an overview. Plus, it has a lot of links to third party crates, to its book (that we will discuss later), and some open source specific stuff.

The docs.rs page of rand is simple. A quick start and then a link of The Rust Rand Book. Before we check out the book, let's first look at the API reference.





The short descriptions is informative that you can differentiate between each type. The difference between `Rng`/`RngCore` seem non-obvious from its name, so the docs spelled it out by indicating that `Rng` is the user-level interface while `RngCore` is the implementation-level interface. Also, the functions' docs do go beyond its name. `random` mentions the thread-local random number generator. `random_ratio`[†] is surprising; the name implies returning a rational

[†] Note that I consider the names as part of the docs.

number (?), but actually it returns a boolean with a customizable weight for `true`. I would've named it `weighted_random_bool`, but that seems too clunky. Ugh, naming is hard.

The traits and modules do have more substance. `Rng` has docs for its usage as a trait bound of generics, while `rand::distr` has a high level view on probability distributions, default distribution used by the crate, and the other distributions that are available.

What I've noticed is that there's not much tutorials and how-to guides on the `docs.rs` page, other than the quickstart example on the main page. Speaking of the main page, let's look at the book linked in it.

The introduction of the book contains a simple table of contents, and external links that concerns this project. Skimming through the book, it shows:

1. A more fleshed out quickstart example
2. The project architecture, its features, platform support, and reproducibility guarantees
3. A comprehensive guide to using random number generators using `rand` (yes, simulating randomness is a deep topic)

4. Migration guides
5. Contributing guidelines

I applaud the main guide and migration guide; These are indepth, well written documentation that gives us the explanation needed for these topics. I like that I can discover the different types of RNGs in the ecosystem, the different distributions available, a topic I didn't know (stochastic processes), fallibility of RNGs, and testing randomness. It discusses the terminologies to beginners, gives a comparison between different generators, and gives notes on its performance and quality. Thus, it feels approachable for someone who has basic competence on what random generators, but has not delved into it deeper. For absolute beginners, I think that the quickstart examples from both `docs.rs` and the rand book is sufficient enough to get started.

Lastly. `rand` is 100% documented according to `docs.rs`. Nice.

`fastrand`

According to the documentation, `fastrand` is a "simple and fast random number generator." It uses Wyrand as its generator.

At first glance, it shows multiple simple examples, its cargo features, and notes on WebAssembly usage.

I don't expect much from `fastrand`. It aims to be simple and easy. I wish it mentions what distribution it uses. It is 100% documented, but the documentation is repetitive. But what can you say? Not much.

`chrono`

The crates.io page discusses the high level overview of its features, including a Wikipedia link to the proleptic Gregorian calendar. Which is nice.

The main page on docs.rs is *meaty*.

Of course.

Time is... erm, *complicated*.

The high level overview from the crates.io page was repeated here. It also adds all the cargo features and their uses. It has an in-depth explanations and examples, grouped in different sections, filled with hyperlinks to the corresponding functions used.

Even they discuss the relation between `chrono` and `time` 0.1!

So in terms of the four quadrants, explanations and how-to guides (via the recipe examples they gave) hold up. Does it have tutorials? Gleaning from the page, I don't think so. Does it *need* a tutorial? It's a library. I don't think there's an idealized end product to made, complete with idealized scenarios. Time is a finicky thing; a library needs to handle any and all potential mishaps and problems that may occur. Thus, the main page consists of multiple how-to guides, detailing the different ways a user can handle certain situations. How do I format my `DateTime`s? The example provides nine ways to do it, each differing in the output.

On the basis of quadrants, we can see that there's a lot of how-to guides and explanations, and we judged that tutorials aren't that necessary. What about the reference?

For starters, we can see that the library is 100% documented. Nice.

The short descriptions are clear enough. `NaiveDate` has the description "ISO 8601 calendar date without timezone. Allows for every proleptic Gregorian date from Jan 1, 262145 BCE to Dec 31, 262143 CE. Also supports the conversion from ISO 8601 ordinal and week date.", which is better than just saying "Timezone-less date", for example. It's complete. It's informative. It even has a link to itself, which is kinda funny.

Clicking on the `NaiveDate` type leads me its docs, which is also extensive. It explains the caveats of the proleptic Gregorian calendar. Then, it explains what a week date is, and how the year number of a week date may not correspond to the actual Gregorian year. Lastly, it discusses the ordinal date as the internal format of `chrono`'s date types.

The subitems of `NaiveDate` are also interesting. `NaiveDate::MIN` and `NaiveDate::MAX` not only state the obvious (that it is the minimum/maximum

possible `NaiveDate`), but also the actual constant[†] . Some of the associated

[†] If you're curious, the minimum possible `NaiveDate` is January 1, 262144 BCE while the maximum possible `NaiveDate` is December 31, 262142 CE.

functions also include the "Errors" and "Panics" sections, which indicates that this project may be using pedantic lints! That's a plus in my book, I just hope they pick and choose instead of blindly doing `#![warn(clippy::pedantic)]`.

From this skim, I'd say it has an okay amount of how-to guides to help the people with basic competencies and some discussions that dig into the history of things. However, word of warning, I'm not particularly well versed in time-related crates and do not know the deficiencies doc-wise without reading the issues on the repo and hearing about testimonials. Nevertheless, at a glance, this seems like a good look. If you, the reader, have found some lapses in the docs, or any docs for the matter, I would love to learn more to improve my heuristics of what makes a certain documentation good or bad.

The docs are somewhat comprehensive in terms of the main section. The discoverability is nice since there's a lot of starting points to go to when needed. It is approachable with lots of how-to-guides and some of its design was moderately explained at the end.

`time`

The crates.io README only has links the docs.rs page, the Time Book, the MSRVP, the contributing guidelines, and the license. Compared to `chrono`, my impressions are that it is a bit sparser as it does not include an explanation to what kind of calendar it uses. Nevertheless, it links to an `mdbook` page, which is a lot more interesting than just putting the explanations and guides in the top-level document.

Before we move on to the book, let's first check out what's going on with the docs.rs page. The page contains a section on its feature flags.. and that's it. I'll comment on the reference later. Let's look at the Time Book instead.

The introduction sells the `time` crate with its features, including being an "easy and safe" crate that is "space optimal and efficient." I'm not going to test these claims, as the goal of this blog post is to have a shallow analysis on the documentation of selected crates. Instead, let's continue on skimming through

the book.

My thoughts:

1. For starters, there are some chapters that don't have any content on it. `mdbook` does have support for draft chapters, so I'm curious why `time` didn't opt into it.
2. There is a section called "How-to guides" and a section called "Technical reference". I wonder if the `time` authors knew about the Diátaxis method.
3. The how-to guides seems not very comprehensive. It only discusses creation and parsing of `Date`/`Time` types. It doesn't seem to discuss about possible unexpected scenarios like non-existent dates or leap seconds or whatever and how to handle them.
4. The technical reference has some explanations, which is what I kinda expected when someone makes a book that contains a "reference" section. It discusses some decisions on API design and why they were made, or just lists out grammar diagrams (which I appreciate at lot).
5. The rest of the book do not have any content in it, so the table of contents kinda misled me into thinking this book was gonna be substantial.

Now we got the Time Book out of the way, let's move on the `docs.rs` page proper. Clicking on the `Date` struct, we see a sparser discussion on what a Date is, and seems like the majority of the method docs only consist of one-to-two sentences description and code snippets full of asserts. I am guessing that they are using the documentation mainly for tests. I'd appreciate enabling the pedantic lints mentioned earlier.

Clicking on the modules honestly makes me disappointed. The module docs consists of a single sentence, and rarely you see a paragraph. The `serde` module has the docs "Differential formats for serde" which is vague, and its submodules only links to the corresponding RFC/ISO formats. I'd appreciate some examples at least of how to use these modules with serde. For what it's worth, `time::serde::format_description!` does have significant docs. My gripe is that it isn't that discoverable, since the time book doesn't have any content on the formatting chapter and the `serde` chapter.

Comparing this to `chrono`, I'm gonna say that `time` has... documentation of lesser quality. The discoverability is not that great, and the reference is lacking. Since there's less content, it is not that approachable. One thing about the Time Book however is that for the pages that are written, at least there were explanation for some design choices of the library.

`jiff`

`jiff` wasn't in the blessed.rs site, but I added it because I remembered that (1) it's a new library by burntsushi[†] , and (2) I was very impressed by its

[†] Known for `regex`, `ripgrep`,`bstr`, and many, many more.

documentation. So I want to revisit that. Now, let's look at its crates.io page.

The README gives an overview, list of hyperlinks for its documentation, an example, a usage tutorial for absolute beginners to Rust, a short fragment on crate features, future plans, performance characteristics, platform support, dependency policy, and its MSRV. That's a lot. I'd say it's *comprehensive*, even just for a README. This gives me hope that the main documentation is as substantial as this.

What's interesting is that in the list of documentation links, there is a bullet point listed as "Comparison with `chrono`, `time`, `hifitime` and `icu`" and "- The API design rationale for Jiff". It's an explicit decision to highlight the *explanation* or discussion of the design choices of `jiff`. Based on earlier crates we have surveyed, this is the first time where you can clearly see and discover that one of the most neglected documentation quadrant is emphasized here. I'll check these links later; first, I want to check the API reference first.

The main page on docs.rs is very, very impressive. I would say at least 90% of it is pure documentation, and I'm not exaggerating. I'm honestly amazed by the amount of effort put in authoring this library's docs. It includes a more comprehensive overview of the library, its features (not the crate features mind you!), the usage taken from the README, hefty amount of short how-to guides with a dedicated paragraph or two (other libraries only have the code snippets and a sentence or two!), and docs on its crate features.

Interestingly, what catches my eye the most is the `_documentation` module, which contains the comparison and design page mentioned before, plus platform support and changelog. Personally I wouldn't have put them in docs.rs, but I imagine that making an `mdbook` project is not worth the effort if you are just looking to dump in your rationalizations somewhere. To be fair, I don't think there would be an "official" way to put in your design decisions anywhere. `mdbook` is typically used for recipes and how-to guides, and tutorials are typically authored by third-party bloggers. The nearest example I could think of are `ARCHITECTURE.md` files, found in some repos, but that's aimed for contributors.

Going back, I want to point the library features here. In `chrono` and `time`, the only thing mentioned that stood out to me was that it was using the proleptic Gregorian calendar. `jiff` does mention using the Gregorian calendar in the not-yet-features section where calendars other than Gregorian is not yet supported. What is amazing though is that `jiff` goes beyond that. It talks about integrating with the IANA Time Zone databases, separation of datetime types, daylight savings, a specific format for formatting and parsing beyond the RFCs/ISOs, etc. It also lists the not-yet-features such as leap seconds, localization, datetime representation, and many more. It's impressive for a library to recognize its flaws and unsupported features then communicate that to the user. Discussions like these implies that there is room for growth.

Going *way* back, I want to take a look at the comparison. Surprisingly, this links me to a markdown file on the Github repo. I noticed that this has the exact same contents as the one in docs.rs, so I checked the source code:

```
/// Longer form documentation for Jiff.
pub mod _documentation {
    #[doc = include_str!("../COMPARE.md")]
    pub mod comparison {}
    #[doc = include_str!("../DESIGN.md")]
    pub mod design {}
    #[doc = include_str!("../PLATFORM.md")]
    pub mod platform {}
    #[doc = include_str!("../CHANGELOG.md")]
    pub mod changelog {}
}
```

Yup. The modules just `include_str!` the markdown files for its docs. I personally like it, as (1) it maintains a single source of truth, and (2) it makes the document discoverable to people who only use `docs.rs` for browsing the documentation and not necessarily using `crates.io` as the starting point. That's an additional point for discoverability.

Moving along, the second paragraph of COMPARISON.md has the statement "The goal of this document is to be as *descriptive* and *substantively complete* as possible," which is a nice way of setting up expectations for your readers. It discusses the differences between `jiff`, `chrono`, `time`, `hifitime`, and `icu`. The author provides certain scenarios where `jiff` performs better (not the speed kind) than other libraries, but there are some scenarios where it falls behind. There is a genuine attempt by the author to be as unbiased as possible. My concern is that while there is justification for the author to create their own library and have a writeup about it, bias may still color the comparisons. Nevertheless, this is a treasure of discussions and gives the user ample understanding of the library and its decisions.

Speaking of, another document that discusses the design decisions is DESIGN.md. This is less about comparisons via code examples or performance characteristics, but rather the philosophy differences between libraries and the rationalization of its API design. The first part are opinionated pieces (remember, from the perspective of the author :p) of comparisons and his experiences with the respective library. I like that he links in a commentary from the original author of the `chrono` crate.

The second part of this file discusses the API decisions of `jiff`. Why are there two duration types? Why isn't there a `TimeZone` trait? Why doesn't `TimeZone` implement `Copy`? And there's more. This is exactly what I look for when I want the *explanations* in the sense of the Diátaxis approach: a document that does not just give me *knowledge*, but also *understanding* of the library. It provides the high-level discussions. It gives context. It is *opinionated*. There is no objective way to write a library. API decisions are colored by the author's decisions, and I *personally* want to know that. I love asking why. Without these explanations, a library's API seem arbitrary and sometimes nonsensical.

After all of that, now let's look at the actual API reference on docs.rs.

I would imagine a beginner user would like to get a `DateTime` or equivalent, but there's no such thing in the top-level module. But by the magic of `Ctrl + F`, we can find that we can use `Zoned` for a `DateTime` with time zones, or `civil::DateTime` for a naïve, imprecise version of this.

Let's check `Zoned` first.

The struct docs has *five* headings.

For comparison, `chrono::DateTime` has two sentences and 31 words, while `time::OffsetDateTime` has two sentences and 13 words. The `jiff` docs confers a degree of complexity for handing time, and rightfully so. It discusses the prominent features of `Zoned`, its caveats, and ways to handle exceptional situations.

Common methods also include substantive amount of docs.`Zoned::new` not only discusses how to construct a `Zoned` struct with `new`, but also with other methods such as `DateTime::in_tz`, `DateTime::to_zoned`, `Date::in_tz`, and `Date::to_zoned`. It also details the problem when converting an ambiguous civil time to zoned datetimes, and solving this via `TimeZone::to_ambiguous_zoned`. It has two "real-world" (well, presented as such) problems like answering the question "What was the civil time in Tasmania at the Unix epoch?" and "What was the civil time in New York when World War 1 ended?"

Impressively, all inherent methods each have at least one paragraph and one example. Skimming this, it has the appropriate "Errors" sections when necessary, and does not repeat itself when needed; opting to linking to related methods whenever necessary. Helpful for discoverability and maintaining a single source of truth!

I want to go on and on with the documentation in other parts of the API reference, but that would be me repeating myself. Instead, I would like to go to the PLATFORM.md and CHANGELOG.md documents to see what's going on. The changelog seems typical; it does not necessarily follow Keep A Changelog's guidelines, but it doesn't deviate very much from a normal changelog.

Documentation on platform support is more substantive than I thought. It has a section on vocabulary[†] , environmental variables, list of platform support

[†] I think this should've been given more emphasis in the main docs.

and some caveats. I can see this being useful for esoteric targets like Android and Wasm, which were discussed here.

All in all, I think the documentation is very comprehensive, pretty discoverable, super approachable, and explains its philosophy pretty well. It has lots of explanations and simple how-to guides, and a comprehensive reference. Just like the previous libraries, I don't think this one would benefit from a hand-holdy tutorial.

`axum`

Moving on from time-related crates, let's look at the web server frameworks. Now, instead of dealing with libraries that allows you to pick and choose your functions and types and use it as you will, frameworks will force you to use it in a certain way just to start working with. I expect that there will be either an entry-point function or type that you must call or construct to start, and your custom functions or types must follow an interface.

With that assumption, let's look at the crates.io page of `axum`. The README discusses its high level features, an example tutorial, performance, and characteristics, MSRV, where to get more examples, help, projects, how to contribute to the framework, and licensing. Pretty typical.

Moving to the `docs.rs` page: it's awesome that it has so many sections in the top-level document. Reminiscent of `jiff`'s docs. Each part of the framework has a minimal discussion on what they are, and it links to its corresponding module for further discussion. Good for discoverability! Examples are sprinkled around the docs too, giving good enough how-to guides for using certain items. There's also docs on the feature flags, which we can appreciate.

Before we dive into the reference, I'd like to look at the examples, showcases, and tutorials linked in the crates.io page. The examples link brings me to its repository in the examples folder. Which is kinda what I expected. The README doesn't provide much info. It doesn't even say what each folder

mean. Clicking on the folders does not give much info, so this tells me that `axum` will just throw me to the deep end and say "You're on your own, kiddo!" Uh oh. At least there are examples here.

Clicking on the showcases hyperlink leads me to a markdown file named ECOSYSTEM.md, specifically on the "Project showcase" heading. It's a bullet list; each point contains a link to the project and a short 1-2 sentence description. Clicking on the tutorials hyperlink leads me to the same file, but on the "Tutorials" heading. Same spiel, but with posts/series from people outside of the tokio group.

Now I'm thinking whether or not should a tutorial be made by the maintainers of the crate, or be delegated to users who wants to teach to other users. But that's probably a topic for another time.

Anyways, I don't really have anything to say for the two hyperlinks here.

Moving on!

Skimming the docs, it seems that the entry point here is `axum::serve`, which uses `Router` and `tokio::new::TcpListener`. I will not discuss the latter type right now, but after looking at `axum::serve`, we should look at `Router` next. The `serve` function has an incredible function signature:

```
pub fn serve<L, M, S>(listener: L, make_service: M) -> Serve<L, M, S>w
    L: Listener,
    M: for<'a> Service<IncomingStream<'a, L>, Error = Infallible, Resp
    S: Service<Request, Response = Response, Error = Infallible> + Clo
    S::Future: Send,
```

Trait soup my beloved.[†]

[†] If you want to learn how to grok these runes and incantations, I recommend watching this video: https://www.youtube.com/watch?v=uh9i3be2wIE

The docs of this function does have a less meaty overview of what it is, and a more meaty section on examples of usage, plus additional links to relevant functions and methods. Interestingly, it has a dedicated section on its return value. I think it's nonstandard though; I probably would've named this section "Errors" instead.

Next, let's look at `Router`.

Oh.

Just a single short description and a caveat to what the generic means. It links to `Router::with_state`, which has a more substantial documentation. It does have lots of examples for handling each caveat, and some explanation on the generic `s`. Honestly, this is one of the best examples (so far) on what a how-to guide should be: discusses how to handle some common scenarios and assumes basic competence on the user.

I guess the documentation is comprehensive, but not to the level of `jiff`. I like the discoverability on the API reference with the amount of linking done, but the examples just makes me feel lost. Approachability, I'd say is fine. The philosophy wasn't discussed that much.

It does have a lot of examples, linked tutorials, good how-to guides in the reference, and of course, 100% documented in docs.rs. That's an "Okay!" in my book.

`actix-web`

`actix-web` is another web server framework (not to be confused with `actix`. That's an actor framework). The crates.io page show the framework features, links to documentations, benchmarks, license, and code of conduct. Not much to say about it, but listing out a partial list of the examples in the "More Examples" section is interesting.

There's objectively less content on the docs.rs page compared to `axum`, but it does link to its website, example repository, and Discord server. It also links relevant types and modules for you to research on, which is a plus point for discoverability.

I'm curious on what the website and its user guide look like, so get on with it. The main page looks sleek; it reminds me of websites I've seen for documentation in other programming languages. It has some snippets for giving the user a taste, which I like.

The book-like docs on their official website delves into the discussion of

`actix-web` as former part of the `actix` ecosystem, and how `actix-web` works. Additional, it has a mini-tutorial and multiple how-to guides. The text looks good, as it discusses how to handle certain scenarios typical for backend server development. It even discusses one pattern not commonly found in other frameworks, which is dependency injection via functions called handlers with magical argument types called extractors.

This site is very comprehensive. It also details advanced topics, protocols, recipes, and patterns using third-party services, and some insights on the framework's HTTP server initialization and connection lifecycle. Surprisingly, it also discusses the `actix` actor framework.

Clearly, the majority of the docs is consolidated in their website. Is it a good idea? Maybe. I would've done a standalone website if:

1. The documentation you want to write does not lend well to the format that crates.io, docs.rs, or mdbook could not.
2. The website includes not just documentation, but also other stuff, like a blog/newsletter, interactive examples, external resources, and others.[†]

   [†] Subtle foreshadowing `:ferrisClueless:`
3. Aesthetic reasons ¯_(ツ)_/¯

The website contents seems mdbook-shaped. It's not really a problem, though; mdbook has a certain aesthetic that the maintainers may not want here. But, it doesn't really have any non-documentation stuff here, so I wonder if they're looking to expanding upon this website more.

Anyways, I digress. Enough pondering. Let's move on to the API reference.

For frameworks, I would like to look into the entry-point item's documentation. In this case, I assume that it will be the `App` struct.

Ah. The doc on the struct is only one sentence. Wait, let's check the percentage of crate documented. Uh oh. 92.09%. I think that's the first time where the library wasn't completely documented. That's sad.

Looking at the methods, I can see that it actually has more substance here. It

at least has multiple sentences and some even has examples! Nice.

The modules continue to be minimal in its documentation. `actix::web` only has bulleted lists with phrases describing the item. Okay. I don't really have much to say here. There *is* documentation. I check the types here and there is more text inside. I just wish I am given the reason why I would use this type over that. At first glance, I can't tell the real differences of `Data`, `ThinData`, and `ReqData` without reading their docs one by one.

Going beyond that, is there a reason for me to check the other modules or is it a matter of "I have a problem and I need to know how to solve it?" If so, is the documentation primed for discoverability? Can I arrive to the answer just from the docs? If I am not familiar with web development, how do I know what search terms should I use here? Hmm, there's a lot of questions to ponder here.

Lastly, I want to check out the examples repository. It's a link to an actual separate repo, woah. The short description states " Curated examples using the Actix ecosystem." Interesting! However, the rest of the document do not discuss what the examples in the repo *do*. You have to guess from the name, which is sad. A minus point for discoverability. There is a lot of examples, though, if you explore the folders.

The remaining part of the document contains a section of a community showcase in the real world, like lemmy, MeiliSearch, and Zero2Prod. Impressive. The next section is titled "Community Articles, Example Apps, Starters & Boilerplate Projects", which is full of tutorials from the community. That's what I've been looking for when it comes to tutorials! For completeness sake, there's also the paid resources section which contains only one bullet point which is the book version of Zero2Prod. Love that for the author of the book, get that bag!

All in all, the amount of tutorials and how-to guides are impressive, but the explanations and references are lacking. It is not that comprehensive, discoverable, and does not explain its philosophy, but it is approachable with the amount of tutorials and examples it has.

`rocket`

Rocket is the third and last backend server framework we will be discussing today. The crates.io page discusses a simple example, links to different parts of its documentation, how to build and test the crate, contribution guidelines, and license. The documentation links are very interesting, as it links to their website instead of an mdbook.

The front page of the website is typical marketing stuff with some snippets showcasing what you can do with `rocket`. I like how the docs in the website is structured. It gives an introduction to the philosophy of `rocket`, a quickstart, some fairly detailed reference pages, a nontrivial tutorial, a conclusion, and a FAQ section. Some differences I've noticed:

1. Rocket *does* discuss on its design philosophy, which I don't remember reading in the docs of both `axum` and `actix-web`.
2. I like the use of book elements such as admonitions here. It emphasizes certain information that may be useful for the user, but does not flow well in the main content.
3. The content is more substantial in the Rocket docs compared to `actix-web`.
4. It actually has a nontrivial tutorial, specifically creating a pastebin. It runs through creating one with Rocket, discusses certain consideration regarding security, and gives a complete program you can run for yourself.

Every essential part of the framework was discussed thoroughly in each section, like requests, responses, state, middleware via fairings, testing, configuration, and deployment. I'll have to say, Rocket's documentation seems more comprehensive compared to `actix-web`, with better ways of handling exceptional situations and showcasing tips by presenting it as admonitions, with its warning, note, and tip callout boxes.

One curious thing about this is that I don't think there's any links to its examples, nor does it have any showcases and third-party tutorials, unlike `actix-web`. It says that the examples are provided in the `examples/` directory, but there's no link to it in the crates.io page. They probably just forgot.

The examples folder does have some good description for each example. A breath of fresh air, finally. I like that there are three complete application examples like `pastebin`, `todo`, and `chat`, and some examples on Rocket's features not discussed much in the main website.

The API reference is interesting, as it is hosted on api.rocket.rs and not docs.rs. Why? I don't know. I just want to point it out. Anyways, the main page shows a usage section, configuration, and testing. There's not much to go around here, but then most of the documentation is in the website, which is fine.

Let's look at the entry level items first, which is the `build` function. Oh no, a single sentence description. Apparently it's just an alias for an associated of the `Rocket` type, so let's look into that instead. Well, there's only two sentences worth of docs here, but it does have a very basic example. The other methods in this page does have more fleshed out examples here, and the main struct have multiple sections on configuration and launching of the application server.

Clicking on the `Config::figment` link has two paragraphs and an example, but just like `Rocket`, the doc on the type itself is where the meat is. It's nice that the fields docs tells you the default values here, which is not very common to see to be quite honest.

My verdict on this crate: very comprehensive and discoverable, with its overall philosophy properly explained in a section. The website is very approachable; the API reference kinda less so. It has a single complete tutorial, many how-to guides, okayish API reference, and a small section for its philosophy and that's about it.

`bevy`

Now, we shall move on to game engines.

Bevy's crates.io page is kinda different from other crates. There's some classics like a list of links to docs, a simple example, licenses, and contributions. However, there's some new stuff, like an explicit **WARNING** section, design goals, an about section, and a community section. That tells

me a lot about the values of this crate's maintainers and in extension, its community. The license is also not like the others, as it lists the caveats of some of its code, not just its general license.

Since I can see a Quick Start Guide link, let's go that first. Ooh, it leads to its own website, just like `actix-web` and `rocket`. The introduction of this guide restates the crate's design goals and its stability warning. The next few pages is a very simple tutorial to create a minimal Bevy app.

I like how some sections are expandable; it's a good way to make the document readable and makes relevant information more accessible. Like, why do I have to scroll around to see where the "Windows" section is when I could just click on it and ignore the rest? It also hides some alternative steps so that the beginner don't get overwhelmed with info. It also explains what certain types do and how it functions in the context of the whole machinery. Terminologies also get explained, albeit only for a brief moment. I appreciate the "Next Steps" section; giving me a list of links directed at beginners so that they can explore more about this framework

I'm still curious about the website, so let's check the front page here. Ah, a classic list of its features. Ooooh, a list of Bevy's financial supporters. Nice.

The "Learn" tab is a list of different documentation links, which is a prime example of discoverability. All guides, tutorials, and references are listed here. Migration guides, contributing guide, bevy assets, and FAQs, oh my!

The examples page is particularly impressive. Numerous pages containing a `<canvas>` you can actually see on your browser and the actual code under it! This is so lovely, I haven't seen anything like it. However, I don't think this method of displaying examples is very much applicable to other crates. But I want to say: if you think you can do it and it makes to do it. please do it and copy Bevy here. Thank you very much.

The contributing guidelines is not just a paragraph here, but a full blown guide just like the quick start guide. It's awesome. Bevy maintainers really values open source principles and fostering a good and helpful community.

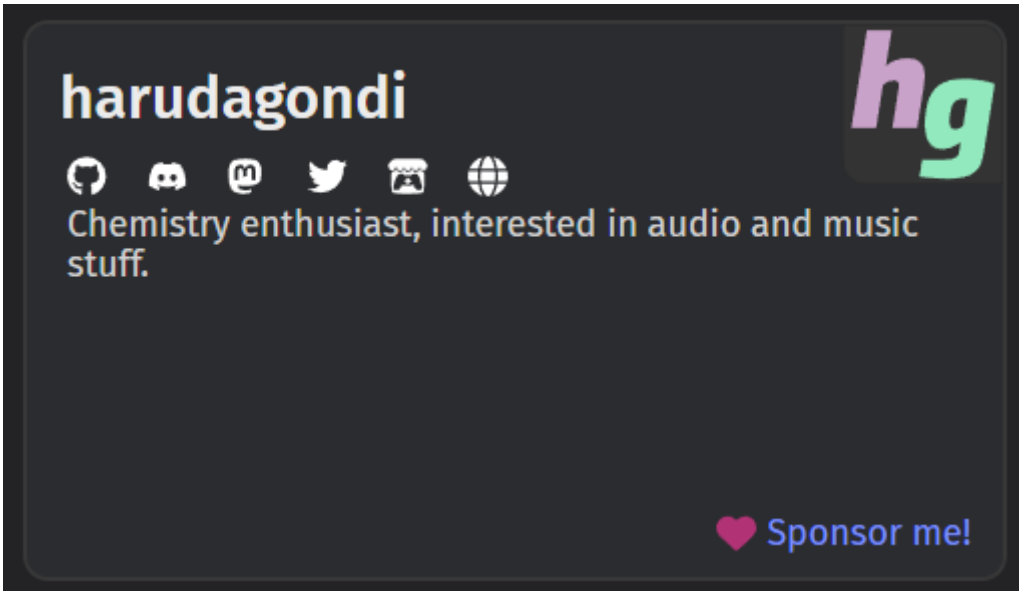
They even have rustc like bevy errors! woag

The frequently asked questions section links me to the GitHub Discussion page on the Bevy repository. I haven't really seen anyone used this feature, so it's interesting that they decide to use this feature.

Bevy Assets shows first-party and third-party tutorials, how-to guides, and external games and tools written with Bevy. It's a step above what `actix-web` do by integrating community posts into its website, and organizing it by topics. I wonder if `actix-web` would be open to adding this kind of thing in their website.

Before we look into the API reference, let's look at the news tab. It's a list of posts, sort of like a newsletter. There's blog posts on new version releases, and some news that is not necessarily code-related, like Bevy being at RustWeek 2025, Bevy Foundation being a public charity, and of course, the most important one: celebrating birthdays. I like it. It kinda humanizes the project; making it seems alive as an open source crate and welcoming for users and contributors.

The community tab is a collection of links for social media accounts, GitHub organization, Q&A, Bevy Assets, and the Bevy People. Clicking on that leads me to a list of people who are part of the bevy organization and look! That's me!



My bad guys. I am actually familiar with Bevy and some previous crates here. To clarify, I am approaching this blog from the perspective who may be new to

these crates, so I just pretend that I don't know any of these. Moving on![†]

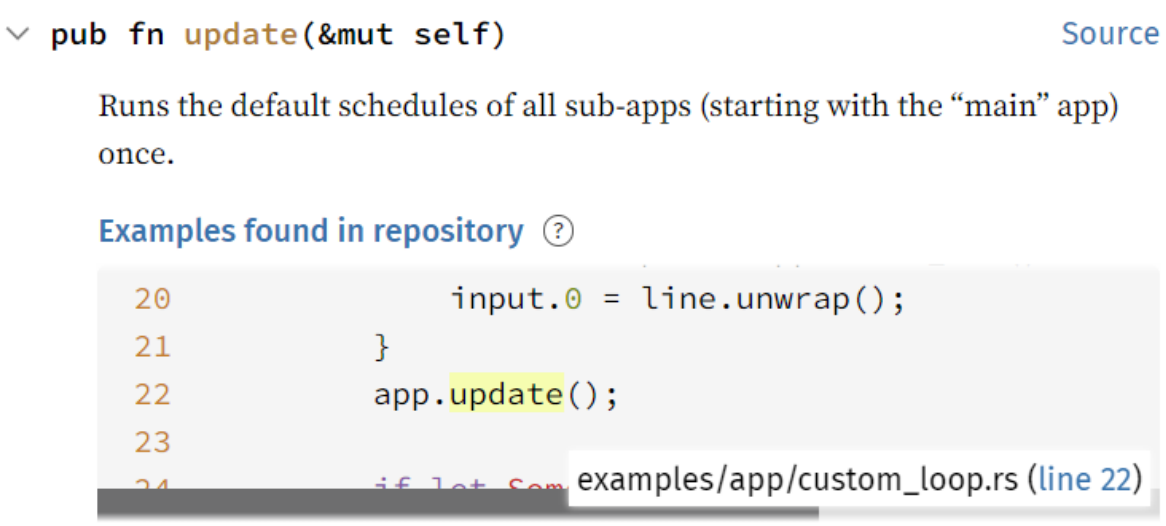
[†] (Anyways, I should really update my Bevy libraries. The past few years have been *pretty* hectic. My bad again guys lol.)

The foundation tab is an overview of the crate's foundation and its model for sustainability of maintaining it. It discusses what exactly is the Bevy Foundation, its leadership, and its transparency processes.

Now the part we are all waiting for: the API reference!

The main page of the docs.rs site contains some links to the website, a "Hello World" example, what this crate is exactly, and the description for each cargo feature. Damn, there is a lot of features. Wow. You have to see it for yourself.

The API reference is okay. I checked the `App` struct and it has two paragraphs of information and an example, and the method docs are as meaty, if not a little bit more meatier. What's interesting is that the docs uses automatically scraped examples from the repository, which is nice.



The other types, however, is much, much more substantial than `App`. `Component` has a *lot*. The type docs consumes like 30% of the page. The method docs? Eh. Only one to two sentences, except for `Component::map_entities`. To be fair, 60% of the page is just the trait's implementors.

`Entity` amusingly has a lot of warnings, notes, and caveats. For such an ubiquitous part of a Bevy program, it's funny that there is a lot of things you

have to consider when using this in a low level way. The trait implementation docs is cute as both `Debug` and `Display` has docs noting the actual format the entity takes form of.

All in all, the documentation is rich in examples, tutorials, and how-to guides, both from the repo itself and from third party posts. It has elaborated docs for community-based stuff such as contribution guidelines, newsletters, and social media/forums/chat , which is nice to see as a way to be approachable to users, especially beginners. The design goals is briefly explained, and the website provides a great way to discover the framework and its examples.

`macroquad`

Unlike Bevy, `macroquad` is a "graphics library", which implies (1) not a framework, (2) not necessarily for games, despite the fact that blessed.rs's description is "a simple and easy to use 2d game library, great for beginners." Anyways, the crates.io page shows us its features, supported platforms, build instructions (which is surprisingly has a lot of docs), community links, and... "async/await". Uh oh. They have continuations at home. It looks kinda janky but the design decision makes (some) sense.

Not much to say here that's different to what I say in previous crates, so let's move on to the API reference—wait is that a link to the website?

```
EOF

xcrun simctl install booted MyGame.app/
xcrun simctl launch booted com.mygame
```

For details and instructions on provisioning for real iphone, check https://macroquad.rs/articles/ios/

▶ Tips

async/await

Why was it not emphasized??? Can't believe that it they did that. Anyways, let's check that site out.

The website does have a simple list of features, and a minimal “Get Started with Macroquad” section, which is nice. Just like Bevy, Macroquad has an example page with playable examples when you click on one. Sadly, unlike Bevy, it doesn’t show its source code, and only links the specific example file to GitHub.

It also has an article tab, which has a list of project updates and guides for the crate. Just like Bevy, I appreciate how the maintainers have a write up on some guides for specific scenarios and some changelog stuff.

What’s interesting is that Macroquad has a “This Week in Quads” page which I don’t think Bevy has, or I have seen yet. Each article details the thoughts of the contributors and what new features are being worked on. Basing on the criteria I’ve set way earlier[†] , it’s one of the best sources for discussing the

[†] *checks notes* uh oh I wrote like 10k words by this point :ferrisClueless:

crate’s philosophy.

The docs tab is an `iframe` to the docs.rs page. What. Let’s just skip that and go to the real docs.rs website.

This page states its supported platforms and its features, plus some minimal example. Thankfully, the `main` macro has some considerable amount of docs, as it includes an error handling part and how to configure it (albeit only shown via code).

The other functions just have a single sentence for docs, like `draw_circle` and `draw_line`. 😕. At least it has scraped examples from the repository.

Going back, I want to click on the Awesome Quads link. That page is a list of game engines built with the `*quad` related libraries, games and apps written with macroquad, posts and publications, example usages, tools, and library integrations. There’s surprisingly a lot, which is nice.

To summarize, the tutorials and how-to guides are provided in the aforementioned “Awesome Quads” link. The website does not detail on creating a Macroquad game, but it does have runnable examples and numerous articles. It gives insight on design decisions via This Week in Quads

and the Articles tabs. Comparing to Bevy, I don't think `macroquad` is more approachable to a beginner in terms of docs, but the code itself seems simple enough.

`fyrox`

Fyrox is an interesting one. Its crates.io page has less content, but has ample links available. Since it's a game engine with an actual scene editor, I'd imagine raw code examples would not be helpful here. Let's check out the Fyrox Book next to see what's going on.

This book is clearly made with mdbook, and looking at the section, it is looking to be very comprehensive. I would like to point this page on design philosophy and goals; it's what I'm really looking for in a crate and it's nice that there's a dedicated page for it. Also, this page on data management has a motivation and technical details section which is useful for people who (1) wants to understand the performance characteristics and (2) the reasoning for Fyrox choosing whatever solution it has right now.

The book is full of screenshots, some even have videos! Skimming through, there's lots of diagrams, explanations, code snippets, a lot of links to the reference, and many more. I don't really want to discuss the details of this book, but I want to say the effort here is commendable. I don't think Bevy really has something comparable to this, but that's probably because of the API instability and constant breaking changes every three months.

Sadly, there's some chapters which are labelled WIP, but then I can't really complain too much given the wealth of information written here. It is reminiscent of the read-the-docs site from Godot, and other game engines with a scene editor such as Unity and Unreal engine.

Wait, there's an official website too that isn't written in the README?? 😞

Okay let's check it out. The website has:

1. A front page with a download link and a play demo projects button.
2. A download page
3. A link to the book (that I don't think link back to the website)

4. Examples that run on the browser
5. A dedicated page for games done in Fyrox
6. A blog that contains update posts, explanations on certain parts of the engine, and some feature articles.
7. A complete list of the game engine's features
8. A list of links to different external crates

Just like the previous gamedev crates, I like the inclusion of examples and having a blog here. The blog posts may not be as flashy and boisterous as Bevy's update posts, but I appreciate it nonetheless. I kinda wish there were more people looking into Fyrox, as Bevy really got the spotlight right now.

Unlike libraries and frameworks such as `actix-web` and Bevy, there's not much information on third party tutorials and how-to guides in this website, nor in the crates.io page, There's some tutorials in the Fyrox book and a single series on game development in Fyrox, but not much else. Considering that this game engine actually has an editor, I would imagine that this type of crate would need the most amount of tutorials, especially blogs with images or video tutorials. However, this is reliant on the Fyrox's community and whether it is "alive" or "energetic" enough. Sad to say, but I honestly think Fyrox could use some marketing like what Bevy does.

Now, I wanna see the API reference next. Uh oh. The docs looks very *minimal*, but it has links to tutorials in the Fyrox book. The docs on the items like `Plugin` are pretty substantial when it has some caveats involved, but other's are kinda missing docs despite the fact that `docs.rs` will tell you that the crate is 100% documented; an example is the methods on this `impl` block on `Handle`. Some items like `Scene` tells me to see the module docs, but then the docs contain only two sentences, which is funny.

To summarize this section, the Fyrox Book contains the majority of documentation of this crate. It is very comprehensive and approachable for competent Rust programmers, but I don't think there's anything aimed to absolute beginners. The Book is also full of how-to guides and high-level reference for the parts of the game engine, with a few tutorials at the end. Consequently, the API reference is incomplete and not that comprehensive,

since I'm assuming that the effort went to the book. The design philosophy is adequately explained in the book and the blog, and the book is primed for allowing the users to discover the extent of this engine, plus linking to the reference for details.

`thiserror`

We shall be moving on to a completely different category of crates: error libraries!

The crates.io of `thiserror` gives an example usage, and... details oooooh. I'd imagine that this would be located in the docs.rs main page but yeah I'm not opposed to this. There's a comparison to anyhow, and that's mostly it. The docs.rs page is exactly the same as the README, so I'm guessing this is just a `include_str!` situation (which is surprisingly... not).

There is exactly one item called `Error`, and that isn't even documented. Honestly, fair. Majority of the docs are in the top level document, anyway.

The details section is comprehensive for what it is, and tutorials/how-to guides aren't really that necessary. There's not really much to say here.

`anyhow`

Because the person who made `thiserror` also made `anyhow`, the documentation looks very, *very* similar. It has a details section just like `thiserror`, and a comparison to the `failure` and `thiserror` crates.

The API reference considerably has more items in it, and the they are well documented. `Error` has a section on display representations. Methods also have some good docs, like it also discusses the caveats of using this struct and some considerations for its API usage.

I see both `thiserror` and `anyhow` as crates simple enough to not need any documentation beyond an API reference. Does it need tutorials and how-to guides? Probably not. Does it need explanations? I would've appreciated it. Is it necessary? I guess not.

`snafu`

The last of the error libraries we will look at is `snafu`. I chose this crate because I've been seeing some increasing popularity in the Rust Programming Language Community Server[†] .

[†] nawt the official rust lang server on the official website btw

The README on the crates.io page is very minimal. It shows a medium-sized example, and links to the documentation and the user's guide. No sections. Hmm, interesting.

I want to check out the user's guide first, so let's click on that. It leads me to docs.rs, but in a module. Kinda like `jiff`! It has links to troubleshooting, examples, design philosophy, showcase of advanced usage via explanations and how-to guides, feature flags docs, compatibility section, upgrading guide, and comparison list. Damn! This is more than what `jiff` has.

I'm gonna try to go through the majority of these to see what's it is all about.

The `troubleshooting` module is apparently a FAQ for the most common issues. Although, there's only one module here for a single error. I guess if people encounter the same problems more frequently (and report them of course), the maintainer may decide to add more modules to this. However, the module docs says:

> *"If you experience a problem and spend a significant amount of time answering it, please consider submitting a pull request to add it here. If you cannot solve your issue after reading this, please open an issue to ask your question."*

This is interesting, I would've imagine that the maintainer would collate all problems and write a module doc for the most common ones. But I guess that this is not the case here. The only submodule here, called `missing_field_source` has a minimal reproducible example, a solution, and most importantly, the *explanation*. I love how it doesn't just end at the solution; it allows the user to understand the *why* of the problem and the solution.

The quick example just links to the main docs.rs page. Uh. Let's skip that.

The “more examples” link goes to the examples module. The docs tells me that looking at the source code for each error type is recommended, and there’s a suggested order to browse the examples. There’s also a note telling me to browse the acceptance and compatibility tests, which “cover a broad range of functionality but with minimal descriptive prose.” Okay, nice. I love the author setting up expectations for me before I click these links.

I will not be checking the backtrace example, only the basic example.

I like the docs here. It explains each context selector, and how to use them via snafu. The source code also shows me a cute little comment telling me that a line isn’t required for most use cases.

basic.rs

```rust
#[derive(Debug, Snafu)]
// This line is only needed to generate documentation; it is not
// needed in most cases:
#[snafu(crate_root(crate), visibility(pub))]
pub enum Error {
    Leaf {
        user_id: i32,
    },

    Intermediate {
        source: std::io::Error,
    },
}
```

Going back a step, the acceptance tests link leads me to a folder in the repository, which uh, doesn’t tell me anything about the files here. But hey! I recognize `basic.rs` here. Let’s click on i—uh, this is not what I saw in the previous example. Uh. Let’s move on to the compatibility tests. Uh oh, it’s like the previous link, but it’s folders now instead of files! And still no README. Sad. Let’s just move on.

Looking at the design philosophy, it gives me a very brief document: with four sections each having 1–2 sentences, this just gives me a feature, use case, and error design recommendation. I was a little bit disappointed. I was kinda expecting something of the caliber of `jiff`’s design rationale.

The `what_code_is_generated` provides an explanation on the expanded derive macro and its design decisions. I like the list of notes on the expanded macro. For the untrained eye, it helps spotting the crucial differences between the original and expanded code. Also some additional code examples I can't grok yet. (My bad, I don't really use this crate)

`opaque` is just a simple how-to guide how to make opaque errors in snafu. `structs` is another how-to guide for using snafu with struct errors. `generics` describes how to use snafu for things with generics, both types and lifetimes. The unique thing about the last one is the "Caveats" section here. Love that.

Ooh, the feature flags module have descriptions that are longer than one sentence. It also states whether or not the feature flag is a default or not. I don't think I've seen people specify that before.

The `compatibility` module, like the feature flags, is a collection of feature flags and their description. The difference is that the feature flags are all about compatibility with older rust versions. I like the "Implies" part, and the description also indicate what specific features this crate uses.

The `upgrading` module is like the migration guides in the previous crates we discussed, specifically the one in Bevy. It has code examples for converting the code from the previous version to the newer version. I really like this kind of stuff, especially if the crate updates relatively frequently.

Lastly, the `comparison` module aims to be an unopinionated document of comparisons of snafu with other libraries. But erm. There's only one submodule here, which is the difference between snafu and `failure`. This page seems like a less elaborated form of `jiff`'s comparison page, which makes it less... comprehensive.

I honestly wish that the author elaborated on their philosophy on error handling more. I wanna know why they created this crate, and what makes their crate special. I would've appreciate both opinion pieces and unbiased comparisons not just with `failure`, but with other modern error handling solutions like `thiserror`, `anyhow`, `error-stack`, `eyre`, `color-eyre`, `miette`, etc. Despite that, I still love the effort of expanding beyond the reference here.

After all of that, now let's move on to the API reference. The docs.rs front page have a list of features with corresponding links, a quick start example, custom error guide, and a small "Next steps" section. I wanna read the entry-point of the library first, so let's check out `Snafu` next.

Wow, compared to `thiserror`, this document has a very significant amount of text in it. Counting the sections, it has... eleven main sections, and 24 sections overall when including subsections. This is incredible.

It even has an attribute cheat sheet formatted as a table!

There's also multiple sections about controlling different things, like `Display`, context, visibility, error sources, backtraces, generated data, stringly-typed errors, and resolution of the snafu crate. The content is extensive and ripe with information, I think you could rely on this for the majority of snafu usage. Each section has at least one paragraph and one example, which is lovely compared to other crates.

Now I'm rethinking about my verdict on `thiserror` and `anyhow`. Can they do more in terms of documentation?

The other items have shorter docs compared to the `Snafu` macro, but still substantial. The `Whatever` struct has a description, an example with a link to the `whatever!` macro, and its limitations. The docs on the macro does elaborate more, with some examples based on the existence of an underlying error.

Overall, the documentation is very comprehensive and approachable, with good discoverability via the amount of hyperlinking in the docs plus the user guide. Numerous how-to guides are available, and the reference is thorough enough. The explanations? Not so much. I want to see the opinions of the author and sell their library to me. What's wrong with other libraries? Why did you make this one?

`clap`

After reviewing the error libraries, let's now review CLI parsers next.

One of the most popular CLI parsers in Rust right now is called `clap`. The

crates.io page. is incredibly minimal. No examples in the README, only a link to docs.rs and examples, plus some sponsors.

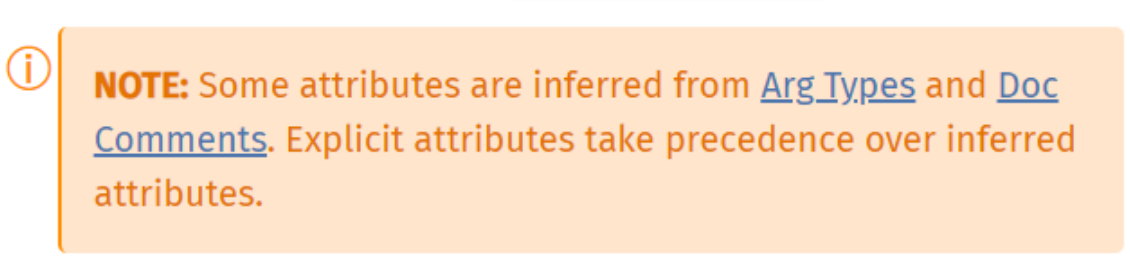
The `docs.rs` is where all the meat is. There's a tutorial and reference for both derive macros and the builder pattern, plus a cookbook, FAQ, discussions page, and a changelog that includes migration guides.

It also has a section on its aspirations, which explains the philosophy and goals of this crate. However, there's not really any links on it beyond shell completions and WG-CLI. I was hoping that there's some more proof here like linking "help generation" to an attribute macro or whatever, or migration guides to port code using other CLI parser libraries to `clap`, and others.

There's a quick example, and a section to its related projects, such as libraries that augment the crate, testing crates, and a CLI app book.

There are modules that contains documentation, just like `jiff` and `snafu`. Let's check each one out first.

The derive reference acts like a mini book, full of sections such as the overview, attributes with its six subsections, argument types, doc comments, mixing the two APIs, and some tips. It has a terminology section, which is nice, and woah:



An admonition? woag

I feel like that when compared to snafu's `Snafu` derive macro, clap's derive tutorial is more organized and concise, due to its bulleted list and tabulated format.

The derive tutorial is very simplistic: it has code examples and then CLI output for each section, with minimal description. Code speaks louder than words, I

guess. At least it doesn't explain too much, or meander to subtopics which detracts the flow of the tutorial. Hmm, is this enough for users?

The builder reference is a dead link 😕

The builder tutorial, just like the derive tutorial, has the same code examples and CLI output. So, my comments on the derive tutorial will also apply here.

The cookbook is a module containing submodules of examples that is either labeled as using the builder API or the derive API. Let's check the `repl_derive` example for fun. Ah. It's just a long code snippet. Mhm. I kinda wanted something like an interactive command line in a website, where you can have a command line on the web, and the example code below just like Bevy.

Wait, let's check if clap has a website that wasn't linked anywhere.

...

Okay there isn't one. Moving on!

`escaped_positional_derive` does at least have outputs here for some sample usage.

The FAQ is one of the most substantial document I've seen so far. It reminds of `jiff`'s comparison and design modules. It has a comparison section and a pitch/anti-pitch, and the rest is what the typical topics seen for a FAQs document. The comparison is eh, as it just compares `clap` to `structopt` and not any other existing CLI parser. 😕

The pitch and anti-pitch is also kinda eh. It has a lot of claims, but not any examples and links to prove the claim. How is it lightweight? How is it "a walk in the park?" In what way is it verbose? What certain features were implemented in a way that is contrary to some use cases?

The discussions link goes to the GitHub's Discussions feature, which is a nice way to provide a forum. Thank god they aren't relying on Discord for forums.

`CHANGELOG.md` is a changelog based on Keep a Changelog. The major versions does have a migration guide, albeit somewhat more minimal than I expected.

Maybe I was just spoiled by Bevy's migration guides.

Now let's check the API reference. Searching for `Parser`, we got the trait and... where's the derive macro? It's not showing up. Now I'll have to check the documentation for `clap_derive`. Oh, the docs for the `Parser` macro is just two sentences. I guess you are really meant to only rely on the derive tutorial as your reference. Kinda weird. It doesn't even describe all the possible usages of each attribute that is available.

The `Parser` trait has some docs, but it seems to be written in a low-level way; not meant to be read by an ordinary user.

Let's check the entry-point item for the builder API, which is the `command!` macro. Erm. A single sentence description and an admonition box. Why does it use `Cargo.toml` and what keys does it read? I don't know, and the builder tutorial doesn't elaborate on it beyond the code snippet.

Is the reference just full of descriptions of limited detail? This is just sad. I was hoping the reference would at least explain every nook and cranny of each part of each item.

To conclude this section, the reference seems not very comprehensive. The discoverability is fine, but I wish they expounded on the examples a bit more so they have an opportunity to add links to the relevant methods. The examples/tutorials/how-to guides are fine, I like them. The philosophy is not very elaborated, but at least it exists.

`pico-args`

Unlike `clap`, `pico-args` does have a longer README, containing a bulleted list of supported and unsupported features, crate feature flags, limitations, and alternatives. There's no code example, however.

True to its name, the crate has only two items. The docs.rs page just repeats a part of the text in the previous page, giving the same bulleted list and the flags.

`Arguments` just has a three word description, but the methods does have better docs here, albeit continuing with the minimal trend. Not only it

describes what it does, but also its warnings and notes. It also includes the recognizable “Errors” heading from the clippy lint, and some scraped examples.

`Error` is... eh. The dataless variants are fine, but the variants with fields are what I have a problem with. At least document what the fields mean.

With the small crate comes with a small summary: fine, but could do better in some areas.

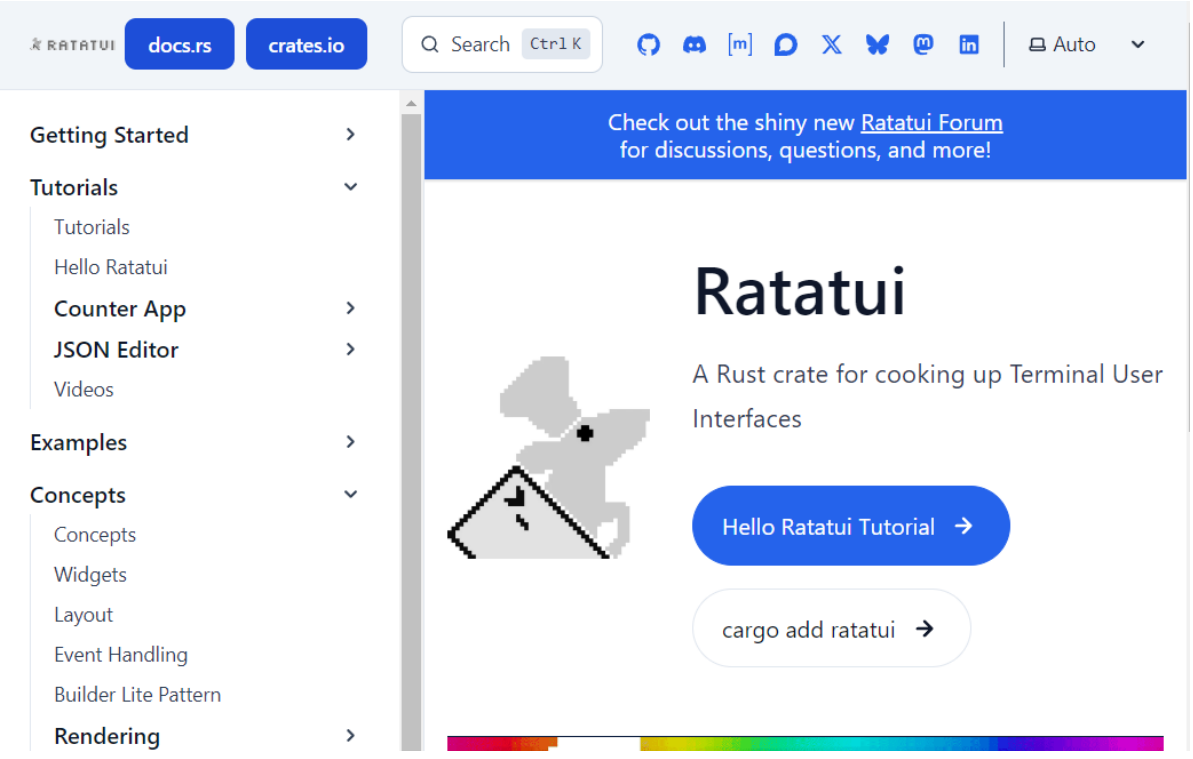
`ratatui`

`ratatui` is not exactly a CLI parser, but rather a TUI. I’m just inserting this crate in this part because the next two crates are all about GUIs.

The README on the crates.io page does have a quick start tutorial, list of links to documentation, an introduction section, layout section, and text plus styling section, contributing guidelines, community projects, and alternative crates.

I also notice that there’s a link to its website at the *top*. That’s awesome. Let’s click on that first.

Wow, the website is a bit... distracting. There’s a sticky header with a lot of buttons, and a navigation sidebar. Like look:



A caveat: I am currently writing this blog post on Obsidian first, with a web viewer on the other side. Also, I have a 14” laptop. So, my view would be quite small.

The front page is a typical marketing page with some selling points why you should use `ratatui`, and some direction to start learning.

Oh my god. There’s gifs in this. And admonitions. And screenshots. And code snippets with file names. And diffs. And expandable code snippets. And adequately detailed paragraphs. And copious amount of pictures. And infographics. And a nontrivial, long tutorial. And external videos on `ratatui`. And an example directory with gifs. And each example has instructions to run the example, a gif, and the code snippet in the page. And there’s actually four different sections on examples. And conceptual explanations. And related literature on concepts. And a peek behind the under-the-hood machineries. And different patterns for the TUI architecture you want. And an overview of terminal backends. And a comparison of backends with a Mermaid diagram. And code recipes. And a substantial FAQs page complete with infographics. And a highlights page akin to Bevy’s update posts. And a showcase of apps and widgets written with `ratatui`. And a page for cargo templates. And the references page is just like an `awesome` repo included in the site. And contributor guidelines for the crate and the website.

Holy shit.

…

The only complaint I have is that they did not link the functions and types used to the API reference. 😔

Other than that, this website is incredible. It’s amazingly comprehensive and

approachable.

The tutorials section shows three tutorials called "Hello Ratatui", "Counter App", and "JSON Editor". Most tutorials in other crates end at the "Hello Ratatui" equivalent, while some end at the counter app. It's nice that `ratatui` is one of those crates that has a nontrivial tutorial with complex moving parts such as the JSON Editor tutorial.

The JSON Editor tutorial details each function needed with a motivational paragraph and the entire snippet; just like a tutorial should. It also guides the user through the process of writing a typical app like what people do when starting a project: making a minimal viable product, and expanding on that piece by piece. I like that when doing the UI part, the tutorial has associated graphics.

The tutorials section even has a list of videos, which is nice.

Just like Bevy, the examples include a link to the example and the example file. The difference is that the example is not interactable, but rather just a gif. I wonder if there's a way to have a terminal backend as a wasm applet.

The concepts section is unique. It discusses the fundamental concepts of this crate, with multiple links of the types to its corresponding docs.rs page. Some types also has some explanations why they exist, such as `WidgetRef` and `StatefulWidgetRef`:

> *"These two traits were introduced in Ratatui 0.26.0 to help avoid a shortcoming that meant that widgets were always consumed on rendering while not breaking all code that has previously been built with that assumption. These two widgets are currently marked as unstable and gated behind the `unstable-widget-ref` feature flag."*

There's also a section on how to use widgets, and how to implement them. I like that this actually explains the code in words. Also, unlike the tutorials, this section exposes the internal machinery of `ratatui` like the "Under the Hood" subsection in Rendering.

Application Patterns is an interesting section. It implies that they're not

endorsing a single architecture for the user to use. Rather, they present these architectures as supplemental reading; quoting:

> *"The documentation on this page is for theoretical understanding and pedagogical purposes only."*

It's not exactly about `ratatui` (as these architectures are crate agnostic), so I appreciate the effort of writing these pages. Also love the diagram (is that UML) here.

Also, the documentation discusses terminal backends here, with a comparison page teaching you how to choose what backend to use.

The recipes act like simple how-to guides, as recipe shows a real-world problem and a solution, plus some different scenarios that could happen/different goals that is needed to be achieved. Collapsing borders? You got it. Creating custom widgets? They have it.

Compared to other FAQs, `ratatui`'s is really nice. It's substantial; it's full of diagrams, images, and gifs, and questions that dip into complex topics have an appropriate external reference for further. Love that they wanna satiate the curiosity of users.

Highlights is less like a changelog and more like a newsletter of updates. Each update is a full blog post that showcases the new features added in each version.

Showcase exhibits the different projects from the users of this crate, with gifs attached. Curiously, there's a showcase of built-in widgets, which in my opinion should probably be a section in the concepts section instead. It's fine. What's more interesting here is the third party widgets, which is nice for discovering new libraries, especially since it includes pics.

Templates is also a unique section that I didn't find in previous crates. Does one need templates for making a `ratatui` app? Well, I'm not gonna question that right now. I'm gonna appreciate the effort instead.

The references page is actually an awesome-like bulleted list of links to

projects and third party crates. Not what I expected for a page named "References" to be honest.

And of course, last but not the least, there's the contributing guidelines part of the documentation, which in this case is actually not a single paragraph! It's super detailed. Makes it super approachable for those who wants to contribute to the project.

Now. After all of that, we shall check out the API reference. The overview is the same as the one in crates.io. I think the entry-point struct here would be the `Terminal`. The documentation is decent with an explanation of how the Terminal works internally. Most methods though are just single sentences description. The most important method though, which is `Terminal::draw` has a considerable amount of docs, even including a list of actions this method will do, and some notes.

The `widgets` module is okay. There's a bulleted list pf built-in widgets with a concise and informative description (at least there's some detail specified that isn't really gleanable from its name). The `Widget` trait docs is neat; there's some historical recounting of how widgets internally worked back then, and some recommendations if you have a niche (?) goal like "widgets immutably work on its data."

From what I can infer, I see the API reference usually includes some explanations, caveats, and some non-obvious behavior of the crate. In the meanwhile, the website clearly aims to be approachable and comprehensive for a beginner to read. There's also some locations where you can discover libraries and examples outside the website.

If I was making a ranking, I would place this at the top to be quite honest. But I'm not 😝 so. Stay tuned for the conclusion! You're almost there!

`egui`

Now, we are starting to discuss about actual GUI libraries, starting with `egui`.

Wow the crates.io page is *long*. It also has a pronounciation guide!

Just like standard crates that has long READMEs, there's an example,

quickstart, features, and FAQs. There's also a section for its design goals and its current development states. Additionally, there's a section describing its dependencies, its purpose and reasoning, and integration possibilities. The README is chockful of explanations and discussions on its design decision, which I didn't expect for this page at all.

I think the "Dependencies" and "Who is equi for?" sections are interesting. There's an emphasis on being a low-dep crate. Also, there's an emphasis on being a niche crate with its own use case. The "Why immediate mode" is unexpectedly long, even providing an advantage/disadvantage subsection. The "Other" section has a subsection on conventions and design choice, which does explain some patterns used here.[†]

[†] Sidenote: I *disagree* that rust should have named and default arguments, but that's an opinion I would love to discuss in another day.

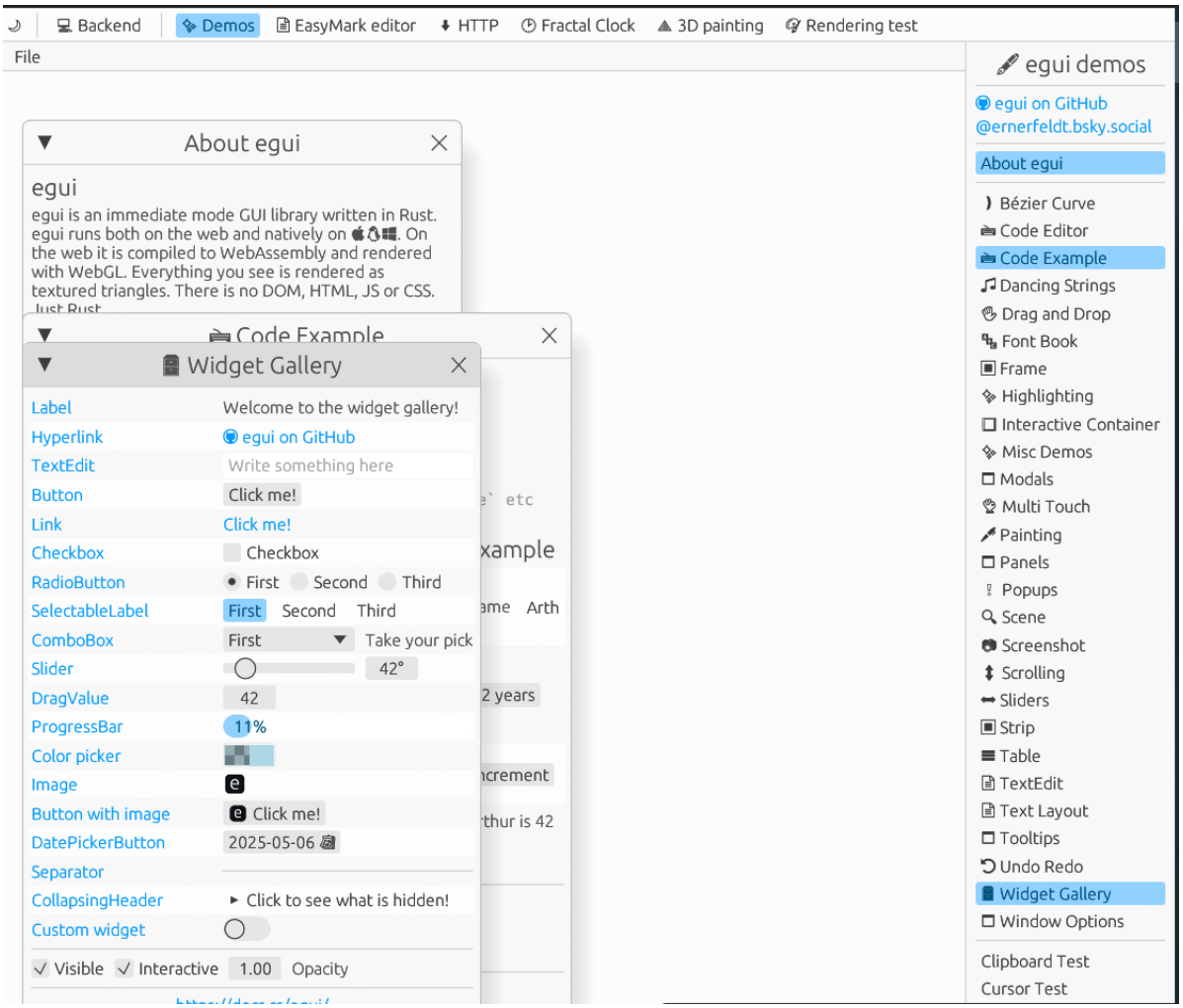
Looking at the docs.rs, there's a small overview, docs of feature flags, a link to the... website. I don't think it was mentioned in the crates.io page, and if it was, it's not emphasized. Before we check out the website, let's continue looking at the main section of docs.rs first. There's multiple examples, a section about viewports and coordinate system, how to integrate egui, how to debug egui, an explanation of immediate mode (which is different from the crates.io page of the advantage/disadvantage), and a miscellaneous section containing some guides.

I like the in-depth discussions on immediate mode and widgets, but I wonder if the main page of docs.rs is the right place for it. Isn't it better if the current miscellaneous page is split into multiple parts like the widgets discussion would be in the `Widget` trait, and the "Auto-sizing panels and windows" part is in the `Layout` struct? I don't know.

My impressions of this page is that it seems disorganized, like why is half of the content in here? Should a beginner of egui "fully grok what immediate mode implies?" Maybe, maybe not. Does it need expand on multi-pass immediate mode in the crate root docs? Probably not. Seems like something that should be in a troubleshooting page or module. It's like a collection of knick-knacks in a document form; yes, they're useful to know, but does a beginner need to know that right now? Should they only discover a section of this when they are debugging, or should they know this upfront?

Now, let's check out the crate's website.

Ah.



It's a demo.

It's a nice demo, to be fair. Each example can be brought up via clicking on the thing at the right side of the page. The examples also have a corresponding link to its source code, which leads me to the specific file in the GitHub repo.

Aside from demos, you can switch to different tabs at the upper portion of the site. The EasyMark editor is a demo of egui's custom markup language. It contains a brief description, design philosophy, details, and feature roadmap. It also have a rendered view of the source code at the right side, which is nice.

The HTTP tab is not clear to me. Is it a demo? Why is it in a tab instead of being part of the demos?

The Fractal Clock, 3D Painting, and Rendering Test tabs seems like demos for rendering in general. The clock is very mesmerizing, I'd have to say. Very effective tool to showcase rendering capabilities.

Unlike other crates that has a website, egui's does not seem to be aimed for people who want to learn via tutorials or how-to guides; rather, it's aimed to those who just wanna see the stuff themselves, and only needs examples. Not much handholding is happening here. I may dock some (nonexistent) points for approachability. (Well I'm not tallying the score anyway, so)

Now for our favorite part of reviewing a crate: the API reference! First, we shall look at the entry-point type of the library, which I think is `Ui`.

The docs contains two "paragraphs" each with one sentence, plus an example code snippet. Okay. I'm assuming the meat of this is in the methods. Well, they aren't exactly faring better, given that they are all mostly just two sentence description, but I'll have to say that I like the links to relevant items in the second sentences.

Most examples I've seen in their repo's examples folder (I don't think this was linked anywhere in the crates.io or docs.rs, but I may have missed it) require `eframe`, which is apparently the official library that allows making apps using `egui` where it actually renders windows to the screen. Which makes me wonder: is there a tutorial or manual on how to do this? I can't tell from the docs I've read.

I don't really know what item to check out next, so uhhhh *throws dart* let's check out the `viewport` module.

Okay! The docs here are measurably better; they're much, much longer. It tells you how to spawn a new viewport and it discusses the different viewport classes it has. It also has a section on viewport usage and integration stuff. It's interesting that there's a short section on future viewport work, linking to a GitHub tracking issue.

Well, to conclude this crate, I guess my general feeling is that despite the crates.io page/main docs.rs page being comprehensive, it feels a bit... disorganized? I don't know if that judgement is fair. I think some parts of the

docs could have been relegated to either the relevant modules/types or the website instead. Speaking of, I wish the site was kinda expanded upon to have a tutorial and some explanations on its design philosophy, like the immediate mode stuff. The crate docs is somewhat less approachable due to the disorganization, but I do like the discoverability in the item docs.

`iced`

The next and last GUI library we will be discussing is iced.

The README on crates.io looks nice; like a typical README of the JavaScript frameworks I've seen, there's centered text and some gifs (yes, the pinnacle of READMEs). The features section is a bulleted list of what `iced` is. There's also an overview section of how its architecture works, accompanied by some mini-tutorial with code snippets. Moreover, there's an intriguing section on its implementation details, which explains some of its design decisions and history.

To be safe, let's check if there's a link explicitly stating that they have a website. They do not. But, the metadata does have the homepage link, leading to iced.rs.

The website is clean, with the front page displaying a typical hero section and showcases of real-world applications using `iced`. The navigation bar is just a drop down menu called "Documentation", with items linking to the iced book, docs.rs page, and a subdomain containing a rustdoc generated reference built from the development branch.

Looking at the iced book, it seems unfinished, with 3 main sections and two sections in the appendix. The written sections however are very approachable, using the 2nd person point of view, simple graphics and diagrams, with a storyteller-like cadence. I like the approach of guiding the reader to build an app from the ground up, and refactoring as we see fit.

I really like the note at the end of the main section. The author tells us that the current form is intended to serve as a quick intro, but admits that the book is not done yet. I'd say its a nice gesture, and illustrates the hard work that comes with documentation. I hope they get the resource and help needed to

finish it, because I really like it.

Moving to the docs.rs page, I'm greeted with a disclaimer. I want to highlight the last two paragraphs here:

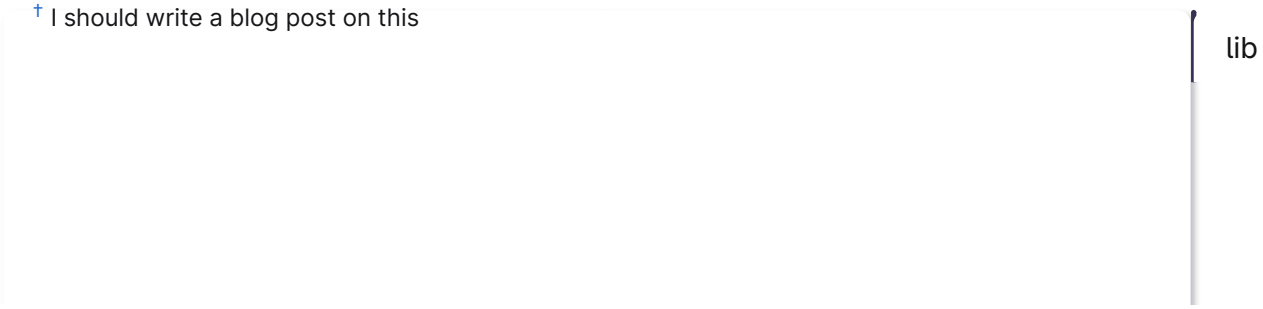
> *"Therefore, iced is easy to learn for **advanced** Rust programmers; but plenty of patient beginners have learned it and had a good time with it. Since it leverages a lot of what Rust has to offer in a type-safe way, it can be a great way to discover Rust itself.*
>
> *If you don't like the sound of that, you expect to be spoonfed, or you feel frustrated and struggle to use the library; then I recommend you to wait patiently until the book is finished."*

Harsh, but I get it. Open source is *hard*, and dedicating the time to documentation is harder when you could be resolving issues or adding new features. However, given that this library is becoming more widely use, where even Pop_OS! is now using iced for their apps, I wonder if there's any interested contributors that would like to improve the iced book. On the other hand, maybe iced breaks too fast with its updates, and you can't really discuss anything beyond the fundamentals without it being out of date very quickly.

The rest of the crate root's docs consists of snippets for each concept of the crate and how to start using them. They aren't an exhaustive explanation of these concepts, but they're a great venue for discovering what iced has to offer here in terms of API. And wow there's a lot of concepts here.

Now let's check the entry point item here, which I'm inferring to be the `application` function. Okay. A single sentence description and a usage example with no explanation. Am I surprised? No. The docs just told me that I'm not gonna be spoonfed. I'm guessing that I'll have to rely on the function signature[†] for documentation. But then look at the signature:

[†] I should write a blog post on this



```
pub fn application<State, Message, Theme, Renderer>(
    title: impl Title<State>,
    update: impl Update<State, Message>,
    view: impl for<'a> View<'a, State, Message, Theme, Renderer>,
) -> Application<impl Program<State = State, Message = Message,
Theme = Theme>>
where
    State: 'static,
    Message: Send + Debug + 'static,
    Theme: Default + DefaultStyle,
    Renderer: Renderer,
```

Well. I personally can understand this. But can a beginner? No, probably not. Can an intermediate to advanced user understand this? Probably yes. Like look: the generics aren't single letter names; they're full words. It's readable if you have the appropriate experience.

I want to check out `Element`, which... okay. It's a type alias where the code block has a horizontal scroll bar. And the docs is just two sentences totalling to fifteen words.

🤷

Yeah, this is gonna be a frustrating experience for a beginner.

At least there's some docs with extensive content, like `iced_core::Element::map` for example.

To sum it all up. I think the iced book is very promising, with its accessible language for beginners, contrasted with its API reference's barebones docs. Is it comprehensive? Eh. The iced book needs not to be, and the reference needs some more examples and explanatory text. The crates.io README does have some of its philosophies written down, and the root docs on docs.rs gives a good overview for the user to discover some of its basic features. Beyond that? Good luck!

`wgpu`

This crate is semi-related to the previous two crates. It's a graphics library!

Yay.

The crates.io page is somewhat long. There's a tabulated links of docs, examples and changelog; and there's a bulleted list of links to all the libraries in the monorepo. The getting started section is intriguing, as it not only have a subsection on Rust, but also on C/C++ and other languages. The community section are links to a Matrix space, which is interesting since most of the crates I've seen before uses Discord for real-time communication. There's also extension specifications. I don't know what that means. There's a table of support platforms, with helpful emojis telling you what kind of support it has, with an accompanying subsection on shader support and whatever "Angle" is. There's even a dedicated section on environment variables and testing.

Checking on the metadata, it does have a homepage, so let's check that out.

Just like `iced`, `wgpu.rs` contains a hero section and a grid of third-party libraries and binaries that uses `wgpu` under the hood. The navigation has links to docs.rs, "download" links (they're just links to crates.io + GitHub repo), and a live demo page that looks kinda scuffed. There's no link to the source code for each example, and the navigation bar seems unCSS'd. Well, at least there's examples, but I don't know how to find the source easily.

Looking at the docs.rs, it just features a small description and feature flags. No simple examples. Probably not possible for a graphics library I guess. The docs does tell me that to start using the API, I should create an `Instance`. So let's check that out first.

At least this has more than two sentences. Kinda eh that the items with minimal docs are very terse. The methods are faring better, like `Instance::new` having a dedicated "Arguments" section for explaining what the argument entails. Too bad some of them are disappoint, like `Instance::from_hal` having the docs "`hal_instance` - wgpu-hal instance."

`wgpu` kinda falls short for me. I appreciate the examples, demos, and the length of the README, but I feel like they could do more here. I guess I was kinda wrong for expecting the docs to be targeted to beginners to `wgpu`.

`tokio`

Now let's move on to async executors, starting with arguably the most popular one: `tokio`!

The crates.io page seems organized and typical of most other crates, like having an overview, examples and link to other guides, contributing section, list of related projects, and other stuff about the crate maintenance itself. Thank god that they have a link to the website that is explicitly labelled as "Website".

Browsing their website, I'm greeted with a typical hero section and a list of sponsors (I'm assuming), some of its high level features, and the `tokio` stack. Remarkably, there's a guide book when clicking on the Learn tab in the navigation bar.

This page contains a tutorial on making a mini-`redis`, applying all the features `tokio` has to offer while guiding you all the way through. Damn, this tutorial is substantive and filled with information about the behavior and some of the idiosyncrasies of its functions and types. After the tutorial proper, there's explanations on how async exactly works under the hood, and discusses some features not yet discussed in the tutorial. Love that for discoverability.

There's also a section called "Topics", which has "self-contained articles related to various topics that come up when writing asynchronous applications." Yep, these are how-to guides. It's interesting that these guides backreference the tutorial on making a "mini-`redis`" which makes me question the "self-contained" aspect of this. There's also two articles named "Getting started with Tracing" and "Next steps with Tracing". I'm getting mixed signals here.

In addition, there's also the glossary. I have not seen that before. It looks good; there's definitely a lot of terminologies with its own connotation in the context of asynchronous programming, so this is useful.

The API documentation tab just links me to the docs.rs page. Let's put a pin on that for later.

Last thing I want to look at in the website is the blog. It basically consists of announcement posts, including both new released versions of the tokio stack

crates and new crates. The latest non-announcement post was back in 2021, which is the first and only dev diary about tokio-console. Sad, I would've like reading more about the thoughts of the developers. Other non-announcement blog posts include an insight on `tower`'s `Service` trait, welcoming someone as the first paid contributor, and how the maintainers reduced latencies with a specific strategy. I would love to read more about the developer's thoughts and tokio's internals. Maybe a "this week/month in tokio" style devlog would be nice?

Unpining the page we pinned earlier, next let's look at the API documentation.

Okay, the crate root docs is at least substantial. Like, it occupies like 90% of the whole page. There's a simple overview, and a tour of tokio which gives a survey of the landscape like its common functions, and modules one should know when working with specific things like async IO and scheduling. The example section is just a code snippet of a simple TCP echo server, with not much explanation. Okay. There's also the standard feature flags docs, which is mostly just "enables X traits/types", and there's unstable feature flags which are API that may break in 1.x.y releases. Lastly, there's a section on support platforms, including `wasm` support, some of its caveats.

Let's check on the entry-point item, which is the `main` attribute macro. The docs are substantial, containing a lot of notes and example usage, not only just the default but also how to use the single threaded runtime, setting the number of worker threads, starting with time paused, handling renamed use imports, and how to handle panics. Nice.

Next, let's check out the `spawn` function, which seems like the bread-and-butter action people would do in an async context. Just like `tokio::main`, the documentation here is long, with a informative description, two examples, when it panics, and how to use `!Send` values from a task. I like that it gives you its guarantees and caveats; not something I see often in these docs.

I'll have to say the docs here is clean, informative, organized, and accessible to beginners and intermediates. The tutorial is comprehensive, the how-to guides are, eh. It's okay. I was hoping for more explanations like the blog posts I pointed out earlier, and the reference isn't an afterthought here.

`smol`

`smol` is another async executor, albeit not as popular as `tokio`.

The README on the crates.io page is considerably shorter than `tokio`'s, but it isn't *nothing*. It has an overview, an example, a list of its subcrates, um, a section about TLS certificates (?), its MSRV policy and license. The crate doesn't have a website, so my only source docs here would be the API reference.

The docs in docs.rs is the same as the first two sections in the README we read earlier, with a link to the examples. Sadly, the example folder doesn't have a README, so it doesn't really makes it clear what the files here mean. What's an `async-h1-client.rs` and `async-h1.server.rs`? I don't know. What's the difference with `simple-client.rs`, `tcp-client.rs`, and `tls-client.rs`? I'd have to click on them.

Let's look at the entry-point item of this crate, which I'm assuming is `block_on`. Ah. A single sentence description and an example. Okay.

`Executor` isn't faring better, with the type docs and most of its methods consists of obvious one liner descriptions and an example. At least `smol` has examples for every part of this. Love the doctests aspect of this. `Executor::spawn_many` is somewhat curious, as it has additional two paragraphs that contrasts this method with the `spawn`, and a note of its behavior when a large number of tasks is spawned.

I don't really have much to say here. I guess I like the amount of examples here.

`embassy`

The third and last async crate we will be looking at is `embassy`.

The crates.io page... is literally just this:

> *"Crate name reserved for the Embassy project: https://github.com/akiles/embassy"*

OK.

The GitHub repo is clearly better, with links to the Embassy Book, the API reference, the website, and the Matrix server. There's also a section on its motivation, its features whose description are actually a paragraph long, and a "sneak peek" which actually is just a quick start example. There's an examples section which explains how the folder is organized, and how to run them (it's not just `` `cargo run --example foo` `` apparently).

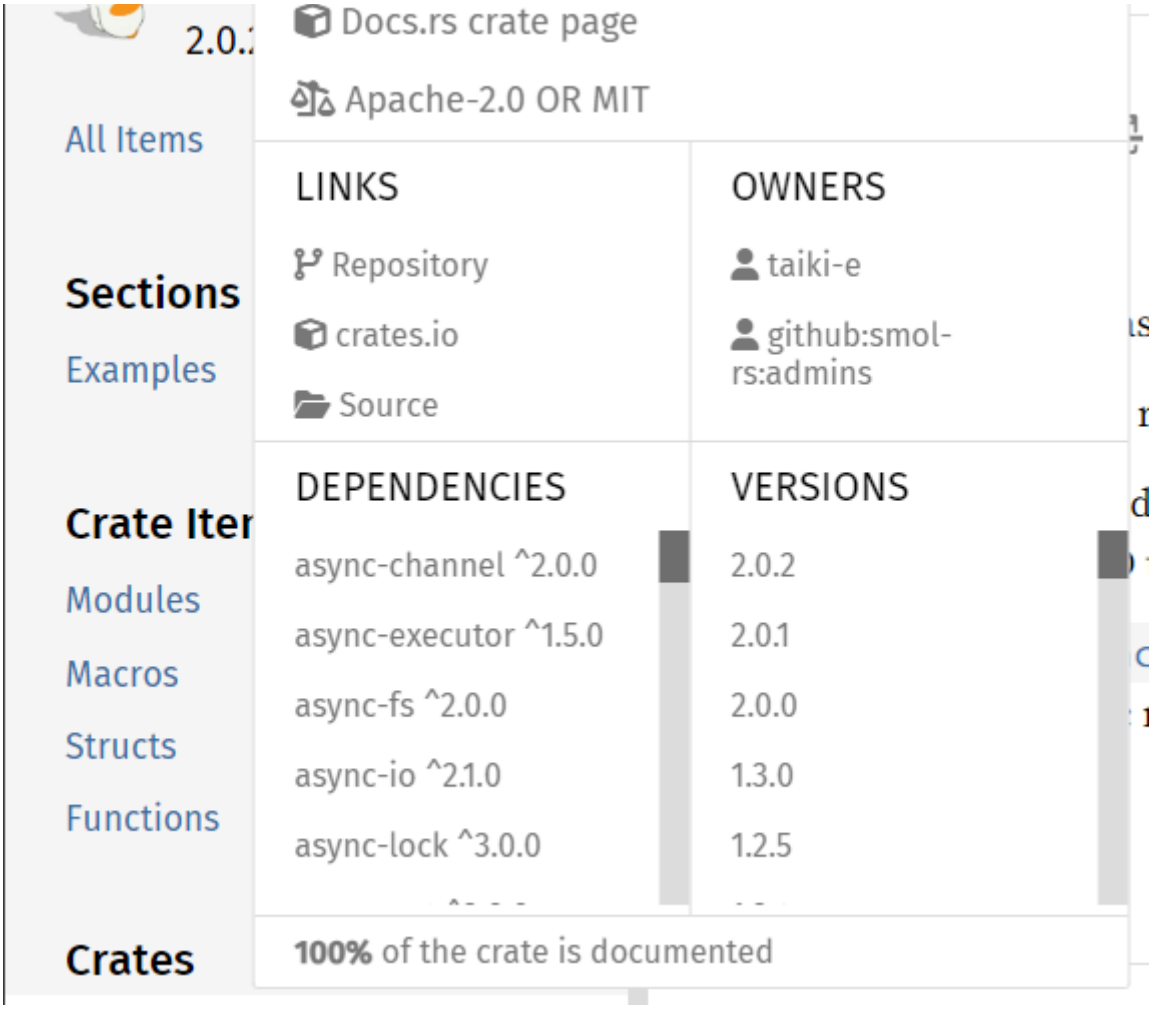
I wanna check out the website first. Ah, just typical hero sections and grid of features. Something new here that I haven't seen yet is the blog, which contains only three posts. The oldest post is literally just titled "Hello World!" and the content is "Hello World!". The other two are announcement posts. Okay. Let's move on.

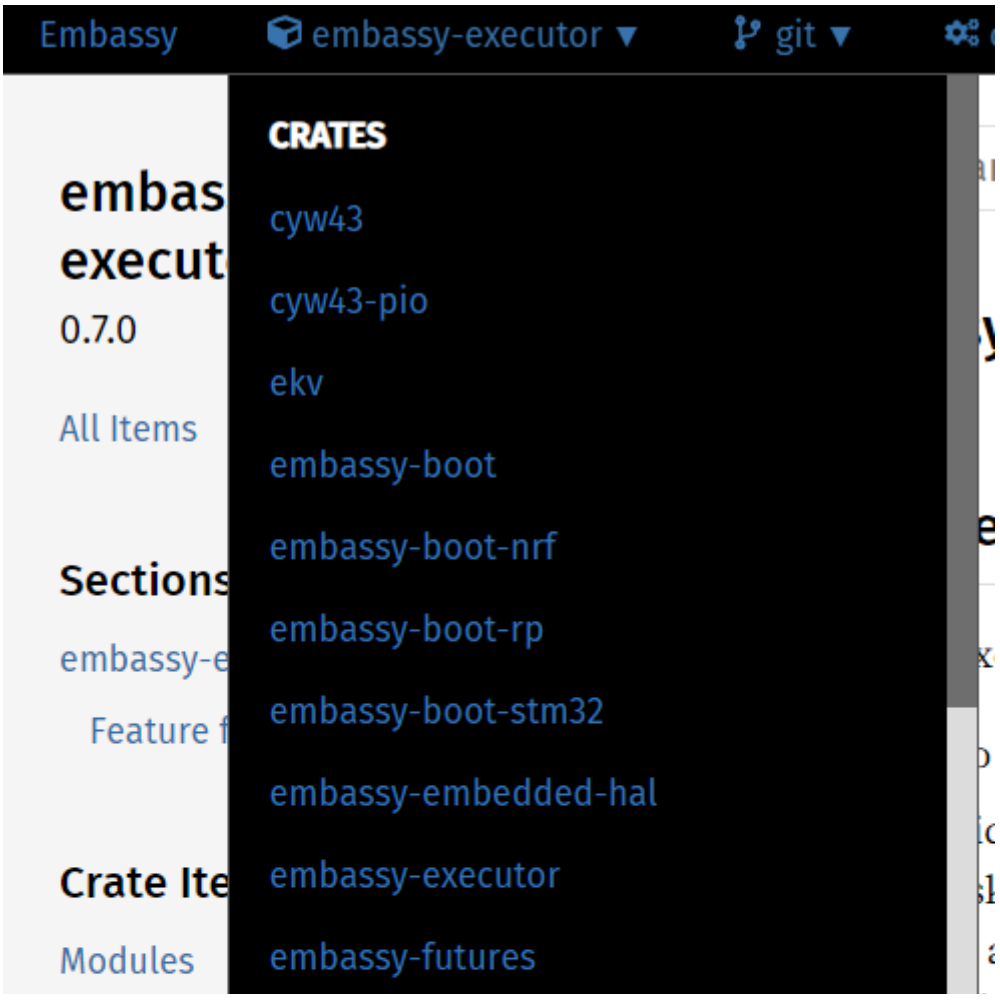
The Embassy Book is just an incredibly long single page. I like that the tutorial anticipates for possible problems like `` `cargo run --release` `` not working. The book also contains a high-level reference and architecture of the crate. There's also a lot of questions in the FAQs section, which is good.

What I appreciate the most here is the "System description" section, which even has diagrams on the executor's behavior and bootloader partitions. It's surprisingly detailed. Also, love the best practices section, haven't seen that before.

Lastly, let's look at the API reference. It links me to the `` `docs.embassy.dev` `` site, which contains its unique behavior and features, and the description of its features flags. Reading the docs of the `` `main` `` macro, it's not as illustrative as tokio's `` `main` `` macro but it does describes its restrictions succinctly. Same could be said for `` `task` ``.

An interesting thing I would like to point out is that unlike docs.rs where the second button in the navbar gives you its links, owners, deps, and versions, embassy's rustdoc generated page has a list of related crates here.


2.0.2
All Items
Sections
Examples
Crate Iter
Modules
Macros
Structs
Functions
Crates
Docs.rs crate page
Apache-2.0 OR MIT
LINKS
Repository
crates.io
Source
OWNERS
taiki-e
github:smol-rs:admins
DEPENDENCIES
async-channel ^2.0.0
async-executor ^1.5.0
async-fs ^2.0.0
async-io ^2.1.0
async-lock ^3.0.0
VERSIONS
2.0.2
2.0.1
2.0.0
1.3.0
1.2.5
100% of the crate is documented


Embassy
embassy-executor ▼
git ▼
embas
execut
0.7.0
All Items
Sections
embassy-e
Feature f
Crate Ite
Modules
CRATES
cyw43
cyw43-pio
ekv
embassy-boot
embassy-boot-nrf
embassy-boot-rp
embassy-boot-stm32
embassy-embedded-hal
embassy-executor
embassy-futures

Seeing the docs on `embassy-time`, I like that there's an extensive overview of how time works especially in the context of embedded device. Sadly, most of the items like `Duration`, `Instant`, and `block_for` have single sentence descriptions.

So, I really like the embassy book, as it is a good starting point, a good source of its design and architecture, and common troubleshooting questions, but the API reference leaves a desire that could have been fleshed out more.

The Rust Official Documentation

Last but not least, let's look at the official docs, including all official books and the standard library.

Of course, we have to start with the main site of the programming language which is https://www.rust-lang.org/. Just like the websites we have seen before, there's a hero section containing a **GET STARTED** button, and a list of its main features. There's also a list of fields and domains Rust excels at, a single paragraph (no links?) about how Rust is used in production, how to get into Rust, and a section dedicated to contributors and foundation members.

Clicking on the "getting started" button leads me to detailed guide on how to install Rust and code in the aforementioned programming language. I like that each part of the guide is divided by the background color, and how there's links to how to setup Rust in some editors. Curiously also includes Helix

(where it links me to the rust-analyzer `mdbook` page?) and Eclipse.[†]

[†] I love the section describing the (unofficial) mascot of Rust, Ferris!

The Learn tab contains links to many of its documentation and tutorials, like the Rust Book, Rustlings, and Rust By Example (RBE). There's also a section on Rust's core documentation, which contains the standard library and five `mdbook`s about its tooling, editions, and compiler errors. Continuing on that, they also link to `mdbook`s on specific application domains such as the command line, WebAssembly, and embedded devices. Lastly, there's also links to more advanced documentation such as the Reference/Informal Specification, the Rustonomicon, and the Unstable Book.

Yup. There's a lot of books.

I would love to discuss every book ever, but that would make this blog post 50,000+ words long, so let's not, probably. 😛

Moving on to other tabs, the Playground link leads me to an interactive code editor where you can run simple Rust programs. Annoying that you can't go back though, even when clicking the previous page button on the browser.

The Tools tab includes links on how to add support for Rust in various editors, just like what we saw in the Getting Start page. There's also some instructions how to use `cargo`, Rust's build tool, and there's a dedicated section to Rust's official tooling for formatting, linting, and documenting.

Next, the Governance page previews how the process of working and improving on the language works. Including the RFC process and all the teams and link to the members and contact details demonstrates the transparency of the Rust team as a whole. (Let's not delve into the details :ferrisClueless:)

Furthermore, the Community tab gives some links to its forums and chat platforms. It's interesting that there's separate forums for users and internals, and they use two different platforms for real time communication. I like that they also include the "This Week in Rust" website, a community-ran newsletter reporting on the new features, crates, and updates that happened in the past week.

Do the governance and community pages count as documentation? I'd argue so. They don't refer to the direct usage of the language, but they do relate to Rust, specifically the social aspects of it. I realized before then that documentation is made by people and meant to be read by people. I know that's obvious, but without recognizing the human and understanding the perspective of others, how can we write documentation that is catered to our users?

My opinion is that the community page is useful because when documentation fails us, we can turn to other people for help. Then, we can diagnose the missing or substandard piece of docs and rewrite it to be better. The governance page is interesting because not only the leadership and its teams decide what features to add, modify, or delete, but also how would they communicate to the multitude of users. With varying degrees of experience and competence in both coding and general communication, the docs must be accessible to all of them, or at least the majority. That's why I like the inclusion of the RFC process here, which gives the power of an ordinary user to suggest changes in an organized way, and some of the aspects the teams consider is teachability to the user, which is critical.

Last in the website that we will discuss is the blog. Okay, we see the classic announcement pages here. But not as much as other crates. There's a lot of other stuff, like reporting security incidents, project goals update, call to actions (maybe a survey?), and a lot of other stuff. I like the monthly project goals update the most, as it gives the reasoning for why they're working for a feature, what happened to that progress, and some help wanted posts and other goal updates not brought to the spotlight.

Before we look at the API reference, I wanna check out a book that is commonly recommended to beginners, nicknamed The Book. This resource is popular as a source recommended to beginners for them to start working with Rust. I'm not going to dive into each chapter here, but I'll give some of my opinions here.

First, this book is not really aimed for absolute beginners to programming in general. It's obvious from the third chapter titled "Common Programming Concepts". This chapter is typically taught in universities for a semester or

two. For example, my high school taught programming in Visual Basic for a year, and we never reached functions. I've also heard of people in IT and engineering courses learning about the basics in a semester. So the fact that these concepts are compressed to a single chapter probably makes it hard for absolute beginners to get over that hump. On the other hand, learning Rust as a first programming language could be very easy as you aren't coming from the mainstream OOP languages and rewiring your brain to work with Rust's paradigm.

My second opinion is that the "Advanced Features" chapter shouldn't be called "advanced." These topics aren't advanced. These are intermediate at best, but I'd consider them part of the fundamentals. You want advanced topics? Read the nomicon. Or the Too Many Lists Book. Or the many blog posts about pointers, provenance, and exposed semantics. I've seen a lot of people skipping this part because they "don't need it," or "they could read it later," which ugh. You are going to encounter that in your second or third mini-project. Let's not waste time not knowing what you don't know.

My third and last opinion is that I wish the Rust Book has a chapter titled "Next Steps". In RPLCS (the community Discord server), I've seen a lot of people asking "what should I do after reading the book?" which is a valid question. Sure, you could make projects, but what if you can't think of one right now. That's why I have made a Carl-bot tag titled `roadmap` which contains several links to resources I have found valuable when learning Rust, and recently refactored it into its own GitHub repository.

Whew. Anyways. Let's move on to what we are we are waiting for: the API reference!

The main page familiarizes you with the landscape and how to navigae them. The section on the tour of the rust standard library lists out containers and collections, and platform abstractions and I/O, which is nice. Let's look at what I think is a representative set of items, which are:

1. `Option`
2. `Vec`
3. `Iterator`

4. `File`
5. `thread::spawn`
6. `std::pin`

The `Option` docs is just "The Option type" and then refers to the module level docs for more information. Okay, let's look at the module. Now this is more promising. It explains the uses of `Option`, interactions with pointers, the try operator, its byte representation, and an overview of its methods. I really love this overview. Just reading the actual type docs seems noiser with the formatting on its functions, but here? It's formatted as bulleted lists with appropriate descriptions that compare and contrasts with othe related methods. I also like the table in the boolean operators section. Very reminiscent of truth tables.

Next, let's check out the `Vec` docs. Unlike the docs on `Option`, the main docs is located on the type proper. There's examples, and sections on indexing and slicing. There's also some explanations on its capacity and reallocation behavior, plus some its guarantees. The guarantees section is very informative. It extensively discusses its memory representation, its behavior in regards to allocation and reallocation via its methods, and drop order.

Some of the methods have very simplistic description, with `Vec::new` having only two sentences and a single line example. I probably would've added using it in `const` and `static`, and maybe mention about the Vec using a dangling pointer. Some methods are actually very educational, especially unsafe functions such as `Vec::from_raw_parts`, with an incredibly long list of invariants that must be followed, and two long examples showcasing this.

The third item we will be looking at is the `Iterator` trait. Just like `Option`, it gives me a very obvious description and a link to the module-level documentation. It has a section on how the module was organized, how Iterator mainly works, how to iterate on stuff, and how to implement it. The documentation also discusses some part on iterator adapters, `Iterator`'s lazy behavior, and the possibility of infinite elements. I like that they pointed out the last three things, but I wish they expounded on the adapter aspect, and how `next` works with nested `Iterator` types. Spoiler alert: unlike JavaScript, `my_iter.map(...).filter(...)` does not mean it maps all elements first, then

filters all resulting elements next.

Fourth, let's look at `File`. It explains what an instance of the struct can do, drop behavior, and caveats on reads and writes. Honestly, the docs here is similar to other crates that has long docs: with several examples, and methods having a medium length of description accompanied by examples. The module docs doesn't really have much to add here.

Fifth, let's check the `thread::spawn` function. Okay, long docs. I like the explanation on its trait bounds. Ooooh, there's more than one code snippet in the examples section, and there's a notes section about unwinding behavior. The module docs explains the threading model of Rust, and an overview of how to spawn threads via the function mentioned earlier. There's also something about thread-local storage, naming threads, and stack sizes, which I am surprised that it wasn't emphasized that much in the `spawn` function.

Last but not least, we should look at the `pin` module, which contains one of the most detailed docs I've ever seen. It explains what "moving" is, what "pinning" is, examples where moving is detrimental, details on pinning, and implement address sensitive types. I think I've only seen the concept of pinning really sublimated in Rust, so it makes sense that the language takes extra effort into explaining this idea.

To conclude this part, the Rust official documentation is the most comprehensive among all crates we've seen before, maximizing approachability and discoverability with the amount of books it has. The blog is not just a glorified announcement page, it includes some explanatory details on current and future features, and some things about the community at whole. However, the majority of the docs is just a big ol' reference, with some guides/tutorials here or there. I guess you can't really have tutorials without getting into very specific domains. And by that point, might as well just browse your favorite crate's docs for that.

Analysis of Crates

Well. I don't expect this section to be very rigorous. More just vibes I guess.

What I noticed in the crates I have reviewed is that there is a wide variety of text content in these documentations, but the mode of delivering these docs are somewhat consistent. There's crates.io, docs.rs, sometimes an mdbook generated document for more complex crates, and a website.

My common criticisms is that API references are typically too short to not explain anything beyond its name. Surely people can say something more than "The Frobnicator struct." What are its exact behavior? Do I have to look at the source code and its unit tests to determine what it does and does not? I think if someone needs to do this, then the documentation failed.

There's some arguments that you could infer the meaning of the function via its signature. *Damn Haskellers*. This is true for some cases, but you generally can't glean the nuance of its behavior just from the signature and their names. This is especially true for unsafe functions, since you can't rely on reading a raw pointer in the signature and make deductions out of that. Unless the crate is sufficient type-safe, relying on signatures 100% isn't feasible.

I've noticed that there's only a few crates that really elaborated on its design philosophy very well. Sure, there are some that lists out its features, but they don't explain why its good and decided to focus on that. Bevy's reasoning on focusing on being data-driven using the ECS paradigm is not really mentioned anyway from what I can tell. There's posts on discussing what makes Bevy's ECS implementation special, but nothing about why ECS over OOP. What motivates the original maintainer to use ECS in the first place? There are pages that discusses the benefits of ECS, but is that what the author was thinking of?

Moving on the topic of websites, some of them kinda fall flat and some are incredibly good. I don't want to name names, because that would be sad and accusatory. But I like it when websites bring something to the table that both the crates.io and docs.rs sites could not. At the very least, I would like to see a main tutorial and multiple how-to guides in these sites, since docs.rs can't do custom sorting of modules without any jank. Does `mdbook` count as a custom website? In this context, I'd say so. However, I am not that familiar with the possible limitations of `mdbook` to say much.

Speaking of tutorials, does dumping a big code snippet to the user counts as

a tutorial? I pointed out these “quick start tutorials” some of these crates have where they only have a single code block that contains a minimal runnable example. Okay great. Now what? I was hoping that we get a step by step process on making a minimal program, narrated by the developer’s thoughts. Maybe even modifying that same program by adding more features! Programming isn’t really a once-and-done deal like what these examples imply. It is an *iterative process*, and I want the tutorials to reflect that.

The benefit of this iterative style, like what Bevy and Ratatui has, is that it lends to more avenues of more organized discoverability and better approachability. Like, if I read a long code snippet, how do I derive its salient functions easily? Compared to a narrative type of tutorials, the author has the choice to emphasize and highlight what functions and types the reader should focus on. This is an important aspect of tutorials in serving the purpose of discoverability. Why waste time reading function docs mentioned in the tutorial when in the real world, it is only really used in tutorials or unit tests? We should consider the reader, not just in the moment where they’re reading the tutorial, but also what happens after that.

When it comes to how-to guides, I think there’s not even much emphasis on this in all the crates I’ve seen here. I guess FAQs act as how-to guides here. Among all the reviewed crates, I like `embassy` the best, as it actually addresses the problem in the perspective of the *user*, and it’s short and sweet. Second would be probably Bevy’s usage of GitHub discussions, since it involves real users, but I don’t know if it is good since the answers are in the perspective of the library author. Same for `clap`’s FAQ page with the perspective stuff, but without the involvement of the user and their problems. Clap’s is mostly about theoretical questions, not practical problems.

My criteria on philosophy seems the most neglected by the majority of the crates. Is it fair to say neglected? I don’t know. Anyways, since the standard library is basically bare bones, and everyone is making their 51st game engine, I think crates would benefit from explaining what sets them from the rest. With that, I really liked `jiff`’s the most. It has the greatest amount of detail written about the rationale of the library, and *why it exists*, as opposed to just contributing to another. Library authors should ask themselves: what makes my library unique? And how do I communicate that to my potential users? It

may help people decide what's best for them, help users understand why the library is the way it is, and help contributors tread through the machinery.

I guess all in all, the popular crates are promising. They *could* do better, and I hope they will. Personally, I believe that documentation is the second most important thing an open source project should be focused on, first being the code of course. Code executes your vision, while documentation makes it accessible. That's why I think internalizing the mantra of "bad docs are bugs" is very essential; programming is not just a mental problem, but also a social one. You can't just publish a docless crate and expect people to use it. Even if your code is a miraculous act of god, if you can't communicate to someone how to use it, then they don't understand it, and they won't use it.

...

Uhhh, I guess I don't really have anything more to say, so to end this section: I want you to repeat "Bad docs are bugs!" three times in front of a mirror to see Ferris give you a high five. Congratulations!

Conclusion

To conclude this 23,000+ words worth a blog post, I wanna start with awarding crates with superlatives I just made up:

1. **Best Design Rationale**: `jiff`
2. **Best Website**: Bevy and Ratatui
3. **Best mdbook**: Fyrox
4. **Best Tutorials**: Rocket
5. **Best How-to Guides**: `embassy`
6. **Best Explanations**: `jiff`
7. **Best API References**: The Rust standard library
8. **Most Comprehensive**: Fyrox or `jiff`, I can't decide.
9. **Most Discoverable**: Bevy

10. **Most Approachable**: Bevy or Rocket, I also can't decide.

I just decided these on the spot, so probably don't think about it too hard.

Anyways, there's a lot of shortcomings with this blog post, so I want to give out recommendations (just like in research papers) for other people to write about this:

1. Approach this review in a quantitative way, somehow.
2. Review non-popular crates.
3. Survey the Rust community on their thoughts on documentation.
4. Actually approach this in rigorous qualitative way. Maybe peruse the Creswell book, I don't know.
5. Write this blog post but better lol

I hope you enjoyed this post and goodbye!

Saturday, May 10th, 2025 — #documentation #ecosystem #programming #rust

---

Other #programming posts:

13 Mar 2026 Torturing rustc by Emulating HKTs, Causing an Inductive Cycle and Borking the Compiler

21 Feb 2026 Parse, don't Validate and Type-Driven Design in Rust

07 Jun 2025 Retrospective (Midyear 2023 – Midyear 2025)

25 Dec 2021 Just rewritten my blog site (again!)

07 Aug 2021 I don't like web development

30 Nov 2020 Moving My Site, and Why I'm Changing Static Site Generators.

Other #rust posts:

13 Mar 2026 Torturing rustc by Emulating HKTs, Causing an Inductive Cycle and Borking the Compiler

21 Feb 2026 Parse, don't Validate and Type-Driven Design in Rust

07 Jun 2025 Retrospective (Midyear 2023 – Midyear 2025)

25 Dec 2021 Just rewritten my blog site (again!)

30 Nov 2020 Moving My Site, and Why I'm Changing Static Site Generators.

Comments

You can reply here!

Just a barren landscape. Here's a flower! 🌺

Sitemap

Tip Jar

Colophon

RSS Feed

Social Media Sites

You can find me on the following links:

Where to Find Me

Mastodon

Webrings

Pinoy Websites: (Previous, Random, Next)

Donation Links

Support me on Ko-fi

LiberaPay

I receive

₱0.00

per week