Collected Fragments

David Heinemeier Hansson and Jason Fried

Table of Contents

On communication

1. You can not not communicate. Not discussing the elephant in the room is communicating. Few things are as important to study, practice, and perfect as clear communication.
2. Real-time sometimes, asynchronous most of the time.
3. Internal communication based on long-form writing, rather than a verbal tradition of meetings, speaking, and chatting, leads to a welcomed reduction in meetings, video conferences, calls, or other real-time opportunities to interrupt and be interrupted.
4. Give meaningful discussions a meaningful amount of time to develop and unfold. Rushing to judgement, or demanding immediate responses, only serves to increase the odds of poor decision making.
5. Meetings are the last resort, not the first option.
6. Writing solidifies, chat dissolves. Substantial decisions start and end with an exchange of complete thoughts, not one-line-at-a-time jousts. If it's important, critical, or fundamental, write it up, don't chat it down.
7. Speaking only helps who's in the room, writing helps everyone. This includes people who couldn't make it, or future employees who join years from now.
8. If your words can be perceived in different ways, they'll be understood in the way which does the most harm.
9. Never expect or require someone to get back to you immediately unless it's a true emergency. The expectation of immediate response is toxic.
10. If you have to repeat yourself, you weren't clear enough the first time. However, if you're talking about something brand new, you may have to repeat yourself for years before you're heard. Pick your repeats wisely.
11. Poor communication creates more work.
12. Companies don't have communication problems, they have miscommunication problems. The smaller the company, group, or team, the fewer opportunities for miscommunication.
13. Five people in a room for an hour isn't a one hour meeting, it's a five hour meeting. Be mindful of the tradeoffs.
14. Be proactive about "wait, what?" questions by providing factual context and spatial context. Factual are the things people also need to know. Spatial is where the communication happens (for example, if it's about a specific to-do, discuss it right under the to-do, not somewhere else).
15. Communication shouldn't require schedule synchronization. Calendars have nothing to do with communication. Writing, rather than speaking or meeting, is independent of schedule and far more direct.
16. "Now" is often the wrong time to say what just popped into your head. It's better to let it filter it through the sieve of time. What's left is the part worth saying.
17. Ask yourself if others will feel compelled to rush their response if you rush your approach.
18. The end of the day has a way of convincing you what you've done is good, but the next morning has a way of telling you the truth. If you aren't sure, sleep on it before saying it.

19. If you want an answer, you have to ask a question. People typically have a lot to say, but they'll volunteer little. Automatic questions on a regular schedule help people practice sharing, writing, and communicating.
20. Occasionally pick random words, sentences, or paragraphs and hit delete. Did it matter?
21. Urgency is overrated, ASAP is poison.
22. If something's going to be difficult to hear or share, invite questions at the end. Ending without the invitation will lead to public silence but private conjecture. This is where rumors breed.
23. Where you put something, and what you call it, matters. When titling something, lead with the most important information. Keep in mind that many technical systems truncate long text or titles.
24. Write at the right time. Sharing something at 5pm may keep someone at work longer. You may have some spare time on a Sunday afternoon to write something, but putting it out there on Sunday may pull people back into work on the weekends. Early Monday morning communication may be buried by other things. There may not be a perfect time, but there's certainly a wrong time. Keep that in mind when you hit send.
25. Great news delivered on the heels of bad news makes both bits worse. The bad news feels like it's being buried, the good news feels like it's being injected to change the mood. Be honest with each by giving them adequate space.
26. Time is on your side, rushing makes conversations worse.
27. Communication is lossy, especially verbal communication. Every hearsay hop adds static and chips at fidelity. Whenever possible, communicate directly with those you're addressing rather than passing the message through intermediaries.
28. Ask if things are clear. Ask what you left out. Ask if there was anything someone was expecting that you didn't cover. Address the gaps before they widen with time.
29. Consider where you put things. The right communication in the wrong place might as well not exist at all. When someone relies on search to find something it's often because it wasn't where they expected something to be.
30. Communication often interrupts, so good communication is often about saying the right thing at the right time in the right way with the fewest side effects.

The Rails doctrine

Ruby on Rails' phenomenal rise to prominence owed much of its lift-off to novel technology and timing. But technological advantages erode over time, and good timing doesn't sustain movements alone over the long term. So a broader explanation of how Rails has continued to not only stay relevant but to grow its impact and community is needed. I propose that the enduring enabler has been and remains its controversial doctrine.

This doctrine has evolved over the past decade, but most of its strongest pillars are also the founding ones. I make no claim to the fundamental originality of these ideas. The chief accomplishment of Rails was to unite and cultivate a strong tribe around a wide set of heretical thoughts about the nature of programming and programmers.

Optimize for programmer happiness

There would be no Rails without Ruby, so it's only fitting that the first doctrinal pillar is lifted straight from the core motivation for creating Ruby.

Ruby's original heresy was indeed to place the happiness of the programmer on a pedestal. Above many other competing and valid concerns that had driven programming languages and ecosystems before it.

Where Python might boast that there's "one and preferably only one way to do something", Ruby relished expressiveness and subtlety. Where Java championed forcefully protecting programmers from themselves, Ruby included a set of sharp knives in the welcome kit. Where Smalltalk drilled a purity of message passing, Ruby accumulated keywords and constructs with an almost gluttonous appetite.

Ruby was different because it valued different things. And most of those things were in service of this yearning for programmer happiness. A pursuit that brought it at odds with not only most other programming environments, but also the mainstream perception of what a programmer was and how they were supposed to act.

Ruby took to not only recognize but accommodate and elevate programmer feelings. Whether they be of inadequacy, whimsy, or joy. Matz jumped implementational hurdles of astounding complexity to make the machine appear to smile at and flatter its human co-conspirator. Ruby is full of optical illusions where that which seems simple, clear, and beautiful to our mind's eye actually is an acrobatic mess of wires under the hood. These choices were not free (ask the JRuby crew about trying to reverse-engineer this magical music box!), which is precisely why they're so commendable.

It was this dedication to an alternate vision for programming and programmers that sealed my love affair with Ruby. It wasn't just ease of use, it wasn't just aesthetics of blocks, it was no one single technical achievement. It was a vision. A counter culture. A place for the misfits of the existing professional programming mold to belong and associate with the like of mind.

I've described this discovery of Ruby in the past as finding a magical glove that just fit my brain perfectly. Better than I had ever imagined any glove could ever fit. But it was even more than that.

It was the event that marked my own personal transition from ‘doing programming because I needed programs’ to ‘doing programming because I fell in love with it as a mode of intellectual exercise and expression’. It was finding a fountain of flow and being able to turn it on at will. For anyone familiar with Csikszentmihalyi’s work, the impact of this is hard to overstate.

I’m not exaggerating when I say that Ruby transformed me and set the course for my life’s work. So deep was the revelation. It imbued me with a calling to do missionary work in service of Matz’s creation. To help spread this profound creation and its benefits.

Now I can imagine most of you shaking your heads with incredulity. I don’t blame you. If someone had described the experience above to me when I was still living under the “programming is just a tool” paradigm, I too would have shook my head. And then I would probably have laughed at the over-the-top use of religious language. But for this to be a truthful account, it also has to be honest, even if that’s off-putting to some or even most.

Anyway, what did this mean for Rails and how does this principle continue to guide its evolution? To answer that, I think it’s instructive to look at another principle that was often used to describe Ruby in the early days: The Principle of Least Surprise. Ruby should behave how you’d expect it to. This is easily described with a contrast to Python:

```
$ irb
irb(main):001:0> exit
$ irb
irb(main):001:0> quit
```

```
$ python
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
```

Ruby accepts both `exit` and `quit` to accommodate the programmer’s obvious desire to quit its interactive console. Python, on the other hand, pedantically instructs the programmer how to properly do what’s requested, even though it obviously knows what is meant (since it’s displaying the error message). That’s a pretty clear-cut, albeit small, example of PoLS.

The reason PoLS fell out of favor with the Ruby community was that this principle is inherently subjective. Least surprising to whom? Well, to Matz. And people who are surprised in the same way as him. As the Ruby community grew, and the ratio of people who were surprised by different things than Matz grew with it, this became a source of fruitless bike-shedding on the mailing lists. So the principle faded to the background, lest to invite more debates going nowhere over whether person X was surprised by behavior Y or not.

So again, what does this have to do with Rails? Well, Rails has been designed with a similar principle to Principle of Least Surprise (To Matz). The Principle of The Bigger Smile (of DHH), which is just what it says on the tin: APIs designed with great attention paid to whatever would make me smile more and broader. When I write it out like this, that sounds almost comically narcissistic, and even I find hard to argue against that first impression.

But creating something like Ruby or Rails is at least at its outset a deeply narcissistic endeavor. Both projects sprung from the mind of a singular creator. But perhaps I'm projecting my own motivations onto Matz here, so let me narrow the scope of my proclamation to that which I know: I created Rails for me. To make me smile, first and foremost. Its utility was to many degrees subservient to its ability to make me enjoy my life more. To enrich my daily toil of wrangling requirements and requests for web information systems.

Like Matz, I at times went to silly lengths to serve my principle. One example is the `Inflector`, a class that understands just enough of the patterns and irregularities of the English language to map a `Person` class to a `People` table, `Analysis` to `Analyses`, and simply `Comment` to `Comments`. This behavior is now accepted as an unquestioned element of Rails, but the fires of controversy raged with great intensity in the early days when we were still coalescing the doctrine and its importance.

Another example that required less implementation effort, but triggered almost as much consternation: `Array#second` through `#fifth` (and `#forty_two` for good trolling measure). These alias accessors were deeply offensive to a very vocal constituency who decried the bloat (and near end of civilization, for good measure) of something that could as well be written as `Array#[1]`, `Array#[2]` (and `Array#[42]`).

But both decisions still, to this day, make me smile. I relish getting to write `people.third` in a test case or the console. No, that's not logical. It's not efficient. It may even be pathological. But it continues to make me smile, thus fulfilling the principle and enriching my life, helping to justify my continued involvement with Rails after 12 years of service.

Unlike, say, optimizing for performance, it's tough to measure optimizing for happiness. This makes it an almost inherently unscientific endeavor, which to some renders it less important, if not outright frustrating. Programmers are taught to argue and conquer the measurable. That which has clear conclusions and where A can categorically be shown to be better than B.

But while the pursuit of happiness is hard to measure at the micro level, it's a lot clearer to observe at the macro level. The Ruby on Rails community is full of people who are here precisely because of this pursuit. They boast of better, more fulfilled working lives. It's in this aggregate of emotions that the victory is clear…

Convention over Configuration

One of the early productivity mottos of Rails went: "You're not a beautiful and unique snowflake". It postulated that by giving up vain individuality, you can leapfrog the toils of mundane decisions, and make faster progress in areas that really matter.

Who cares what format your database primary keys are described by? Does it really matter whether it's "id", "postId", "posts_id", or "pid"? Is this a decision that's worthy of recurrent deliberation? No.

Part of the Rails' mission is to swing its machete at the thick, and ever growing, jungle of recurring decisions that face developers creating information systems for the web. There are thousands of such decisions that just need to be made once, and if someone else can do it for you, all the better.

Not only does the transfer of configuration to convention free us from deliberation, it also provides a lush field to grow deeper abstractions. If we can depend on a `Person` class mapping to `people` table, we can use that same inflection to map an association declared as `has_many :people` to look for a `Person` class. The power of good conventions is that they pay dividends across a wide spectrum of use.

But beyond the productivity gains for experts, conventions also lower the barriers of entry for beginners. There are so many conventions in Rails that a beginner doesn't even need to know about, but can just benefit from in ignorance. It's possible to create great applications without knowing why everything is the way it is.

That's not possible if your framework is merely a thick textbook and your new application a blank piece of paper. It takes immense effort to even figure out where and how to start. Half the battle of getting going is finding a thread to pull.

The same goes even when you understand how all the pieces go together. When there's an obvious next step for every change, we can scoot through the many parts of an application that is the same or very similar to all the other applications that went before it. A place for everything and everything in its place. Constraints liberate even the most able minds.

As with anything, though, the power of convention isn't without peril. When Rails makes it so trivial to do so much, it is easy to think every aspect of an application can be formed by precut templates. But most applications worth building have some elements that are unique in some way. It may only be 5% or 1%, but it's there.

The hard part is knowing when to stray from convention. When are the deviating particulars grave enough to warrant an excursion? I contend that most impulses to be a beautiful and unique snowflake are ill considered, and that the cost of going off the Rails is under appreciated, but just enough of them won't be that you need to examine all of them carefully.

The menu is omakase

How do you know what to order in a restaurant when you don't know what's good? Well, if you let the chef choose, you can probably assume a good meal, even before you know what "good" is. That is omakase. A way to eat well that requires you neither be an expert in the cuisine nor blessed with blind luck at picking in the dark.

For programming, the benefits of this practice, letting others assemble your stack, is similar to those we derive from Convention over Configuration, but at a higher level. Where CoC is occupied with how we best use individual frameworks, omakase is concerned with which frameworks, and how they fit together.

This is at odds with the revered programming tradition of presenting available tools as individual choices, and to bestow the individual programmer privilege (and burden!) of deciding.

You've surely heard, and probably nodded to, "use the best tool for the job". It sounds so elementary as to be beyond debate, but being able to pick the "best tool" depends on a foundation that allows "best" to be determined with confidence. This is much harder than it seems.

It is a problem similar to that of the diner in a restaurant. And like picking each course in an eightset meal, picking each individual library or framework is not a job done in isolation. The objective in both cases is to consider the whole evening or system.

So with Rails we decided to diminish one good, a programmer's individual privilege to choose each tool in their box, for a greater one: A better tool box for all. The dividends are legion:

1. **There's safety in numbers**: When most people are using Rails in the same default ways, we have a shared experience. This common ground makes it much easier to teach and help people. It lays a foundation for debate on approach. We all watched the same show last night at 7, so we can talk about it the next day. It fosters a stronger sense of community.
2. **People are perfecting the same, basic tool box**: As a full-stack framework, Rails has a lot of moving parts, and how those work together is as important as what they do in isolation. Much of the pain in software comes not from the individual components, but from their interaction. When we all work on alleviating shared pain from components that are configured and fail in the same ways, then we all experience less pain.
3. **Substitutions are still possible, but not required**: While Rails is an omakase stack, it still allows you to replace certain frameworks or libraries with alternatives. It just doesn't require you to. Which means you can delay those decisions until you've developed a clear, personal palate that may prefer the occasional difference.

Because even the most learned and skilled programmers who come to and stay in Rails aren't likely opposed to all matters of the menu. (If they were, they probably wouldn't have stuck with Rails.) So they pick their substitutions with diligence, and then go on to enjoy the rest of the curated, shared stack alongside everyone else.

No one paradigm

There's a strong emotional appeal to picking a single central idea and following it to the logical conclusion as your architectural underpinning. There's a purity in such discipline, so it's clear why programmers are naturally attracted to this bright light.

Rails isn't like that. It isn't a single, perfect cut of cloth. It's a quilt. A composite of many different ideas and even paradigms. Many that would usually be seen in conflict, if contrasted alone and one by one. But that's not what we're trying to do. It isn't a single championship of superior ideas where a sole winner must be declared.

Take the templates we build the view in our Rails MVC pie with. By default, all the helpers that allow us to extract code from these templates are just a big pot of functions! It's a single namespace even. Oh the shock and the horror, it's like PHP soup!

But I contend that PHP had it right when it came to presenting individual functions that rarely needed to interact, as is the case with much abstraction in view templates. And for this purpose, the single namespace, the big pot of methods, is not only a reasonable choice, but a great one.

This doesn't mean we don't occasionally want to reach for something more object-oriented when building views. The concept of `Presenters`, where we wrap many methods that are interdependent with each other and the data below it, can occasionally be the perfect antidote to a soup of methods turned sour by dependencies. But it's generally proved to be the rare rather than common fit.

In comparison, we generally treat the model in our MVC layer cake as the prime bastion of object-oriented goodness. Finding just the right names for objects, increasing the coherence, and lowering the coupling is the fun of domain modeling. It's a very different layer from the view, so we take a different approach.

But even here we don't subscribe to single-paradigm dogma. Rails concerns, the specialization of Ruby's mixins, are often used to give the individual models a very wide surface area. This fits well with the Active Record pattern by giving the concerned methods direct access to the data and storage they interact with.

Even the very foundation of the Active Record framework offends some purists. We're mixing the logic needed for interfacing with the database directly with the business domain and logic. Such conflation of boundaries! Yes, because it proved to be the practical way to skin a web-app cat that virtually always talks to a database of some sort to persist the state of the domain model.

To be so ideologically flexible is what enables Rails to tackle such a wide array of problems. Most individual paradigms do very well within a certain slice of the problem space, but become awkward or rigid when applied beyond its natural sphere of comfort. By applying many overlapping paradigms, we cover the flanks and guard the rear. The final framework is far stronger and more capable than any individual paradigm would have allowed it to be.

Now, the cost of this polyamorous relationship with the many paradigms of programming is conceptual overhead. It's not enough to just know object-oriented programming to have a good time with Rails. It's preferable to be well served with procedural and functional experiences as well.

This applies to the many sub-languages of Rails as well. We don't try to shield you that much from having to learn, say, JavaScript for the view or SQL for the occasional complicated query. At least not to reach the peaks of possibilities.

The way to alleviate some of that learning burden is to simply just make it easy to get started, make something of real value, before you understand every single aspect of the framework. We have a rush to Hello World for this reason. Your table already prepared and an appetizer served.

The thinking is that by giving something of real value early, we'll encourage the practitioners of Rails to level-up quickly. Accept their journey of learning as a joy, not an obstacle.

Exalt beautiful code

We write code not just to be understood by the computer or other programmers, but to bask in the warm glow of beauty. Aesthetically pleasing code is a value unto itself and should be pursued with vigor. That doesn't mean that beautiful code always trumps other concerns, but it should have a full seat at the table of priorities.

So what is beautiful code? In Ruby, it's often somewhere at the intersection between native Ruby idioms and the power of a custom domain-specific language. It's a fuzzy line, but one well worth trying to dance.

Here's a simple example from Active Record:

```
class Project < ApplicationRecord
  belongs_to :account
  has_many :participants, class_name: 'Person'
  validates_presence_of :name
end
```

This looks like DSL, but it's really just a class definition with three class-method calls that take symbols and options. There's nothing fancy here. But it sure is pretty. It sure is simple. It gives an immense amount of power and flexibility from those few declarations.

Part of the beauty comes from these calls honoring the previous principles, like Convention over Configuration. When we call `belongs_to :account`, we're assuming that the foreign key is called `account_id` and that it lives in the `projects` table. When we have to designate the `class_name` of `Person` to the role of the `participants` association, we require just that class name definition. From it we'll derive, again, the foreign keys and other configuration points.

Here's another example from the database migrations system:

```
class CreateAccounts < ActiveRecord::Migration
  def change
    create_table :accounts do |t|
      t.integer :queenbee_id
      t.timestamps
    end
  end
end
```

This is the essence of framework power. The programmer declares a class according to certain convention, like a `ActiveRecord::Migration` subclass that implements `#change`, and the framework can do all the plumbing that goes around that, and know this is the method to call.

This leaves the programmer with very little code to write. In the case of migrations, not only will this allow a call to `rails db:migrate` to upgrade the database to add this new table, it'll also allow it to go the other way of dropping this table with another call. This is very different from a programmer making all this happen and stitching the workflow together from libraries they call themselves.

Sometimes beautiful code is more subtle, though. It's less about making something as short or powerful as possible, but more about making the rhythm of the declaration flow.

These two statements do the same:

```
if people.include? person
```

```
if person.in? people
```

But the flow and focus is subtly different. In the first statement, the focus is on the collection. That's our subject. In the second statement, the subject is clearly the person. There's not much between the two statements in length, but I'll contend that the second is far more beautiful and likely to make me smile when used in a spot where the condition is about the person.

Provide sharp knives

Ruby includes a lot of sharp knives in its drawer of features. Not by accident, but by design. The most famous is monkey patching: The power to change existing classes and methods.

This power has frequently been derided as simply too much for mere mortal programmers to handle. People from more restrictive environments used to imagine all sorts of calamities that would doom Ruby because of the immense trust the language showed its speakers with this feature.

If you can change anything, what is there to stop you from overwriting `String#capitalize` so that `"something bold".capitalize` returns "Something Bold" rather than "Something bold"? That might work in your local application, but then break all sorts of auxiliary code that depend on the original implementation.

Nothing, is the answer. There's nothing programmatically in Ruby to stop you using its sharp knives to cut ties with reason. We enforce such good senses by convention, by nudges, and through education. Not by banning sharp knives from the kitchen and insisting everyone use spoons to slice tomatoes.

Because the flip side of monkey patching is the power to do such feats of wonder as `2.days.ago` (which returns a date two days back from the current). Now you might well think that's a bad trade. That you'd rather lose `2.days.ago` if it means preventing programmers from overwriting `String#capitalize`. If that's your position, Ruby is probably not for you.

Yet it'd be hard — even for people who would give up such freedom for some security — to argue that the power to change core classes and methods has doomed Ruby as a language. On the contrary, the language flourished exactly because it offered a different and radical perspective on the role of the programmer: That they could be trusted with sharp knives.

And not only trusted, but taught in the ways to use such capable tools. That we could elevate the entire profession by assuming most programmers would want to become better programmers, capable of wielding sharp knives without cutting off their fingers. That's an incredibly aspirational idea, and one that runs counter to a lot of programmer's intuition about other programmers.

Because it's always about other programmers when the value of sharp knives is contested. I've yet to hear a single programmer put up their hand and say "I can't trust myself with this power, please take it away from me!" It's always "I think other programmers would abuse this". That line of paternalism has never appealed to me…

Value integrated systems

Rails can be used in many contexts, but its first love is the making of integrated systems: Majestic monoliths! A whole system that addresses an entire problem. This means Rails is concerned with everything from the front-end JavaScript needed to make live updates to how the database is migrated from one version to another in production.

That’s a very broad scope, as we’ve discussed, but no broader than to be realistic to understand for a single person. Rails specifically seeks to equip generalist individuals to make these full systems. Its purpose is not to segregate specialists into small niches and then require whole teams of such in order to build anything of enduring value.

It is this focus on empowering the individual that points to the integrated system. It’s in the integrated system we can cut out many needless abstractions, reduce the duplication between layers (like templates on both the server and the client), and, above all, avoid distributing our system before we absolutely, positively have to.

Much of the complication in systems development comes from introducing new boundaries between the elements that restrict how you make calls between A and B. Method calls between objects is far simpler than remote procedure calls between microservices. There’s a whole new world of hurt in failure states, latency issues, and dependency update schedules that await those who venture into the lair of distribution.

Sometimes this distribution is simply necessary. If you want to create an API to your web application that other people can call over HTTP, well, then you just have to suck it up and deal with many of these issues (although handling requests inbound rather than sending them outbound is much easier — your downtime is someone else’s failure state!). But that’s at least a limited amount of damage inflicted on your own personal development experience.

What’s worse is when systems are prematurely disintegrated and broken into services or, even worse, microservices. This drive frequently starts from the misconception that if you want a Modern Internet Application, you’ll simply have to build the systems many times over: Once on the server side, once on the JavaScript MVC client-side, once for each of the native mobile applications, and so forth. This is not a law of nature, it needn’t be so.

It’s entirely possible to share large chunks of the entire application across multiple apps and accesses. To use the same controllers and views for the desktop web as for embedded in native mobile apps. To centralize as much as possible within that glorious, majestic monolith: The integrated system.

All this without giving up much if anything in terms of speed, user experience, or other attributes that falsely draw developers to premature distribution.

That’s the have-most-of-it-all we seek: All the power of individually tuned and distributed applications with the ease-of-use and understanding of a single, integrated system.

Progress over stability

When systems have been around for more than a decade, like Rails has, their natural tendency is towards ossification. There are a million reasons why every change might be an issue for someone, somewhere who depended on past behavior. And fair reasons those are too, for the individual.

But if we listen too closely to the voices of conservatism, we’ll never see what’s on the other side. We have to dare occasionally break and change how things are to evolve and grow. It is this evolution that'll keep Rails fit for survival and prosperity in the decade(s?) to come.

This is all easy to understand in theory, but much harder to swallow in practice. Especially when it's your application that breaks from a backwards-incompatible change in a major version of Rails. It's at those times we need to remember this value, that we cherish progress over stability, to give us the strength to debug the busted, figure it out, and move with the times…

Progress is ultimately mostly about people and their willingness to push change. This is why there are no lifetime seats in groups like Rails Core or Rails Committers. Both groups are for those who are actively working on making progress for the framework. For some, their stake in such progress may last just a few years, and we will forever be grateful for their service, and for others it may last decades.

Likewise, it's why it's so important for us to continue to welcome and encourage new members of the community. We need fresh blood and fresh ideas to make better progress.

Push up a big tent

With so many controversial ideas to its credit, Rails could quickly become an insular group of ideological hermits, if we required everyone to exhibit complete deference to all tenets, all the time. So we don't!

We need disagreement. We need dialects. We need diversity of thought and people. It's in this melting pot of ideas we'll get the best commons for all to share. Lots of people chipping in their two cents, in code or considered argument…

Having a big tent doesn't mean trying to be all things to all people, though. It just means you welcome all people to your party, and allow them to bring their own drinks. We need to lose none of our soul or values by offering others to join us, and we may well learn how to mix a new delicious drink or two.

This doesn't come for free. It requires work to be welcoming. Especially if your goal isn't just to attract more people who are just like the ones who are already part of the community. Lowering the barriers to entry is work we should always take seriously.

You never know when the next person who starts just fixing a misspelling in the documentation ends up implementing the next great feature. But you stand a chance to find out if you smile and say thank you for whatever small contribution that gets the motivation flowing.