Go is the new Ruby

How I fell in love with Ruby

I discovered Ruby around the time Ruby on Rails was announced.

At that time, PHP was ubiquitous for web development, even though great frameworks didn't really exist yet. But Rails was a game changer. Convention over configuration was amazing. Name things consistently, and the framework will automatically figure out how to make all the parts of your application access these things, without having to write any glue! Building websites using Rails was a satisfying experience. I also enjoyed the fact that the framework would take care of embracing the best practices, so I can focus on just describing the logic rather than implementation details.

But what I enjoyed even more than Rails was Ruby. To people implementing Ruby interpreters and compilers, Ruby's grammar is atrocious. But as a developer, Ruby is a pleasure to use. Ruby is an extremely flexible language. There are many different ways to do something. This can be a weakness if abused (monkey-patching can turn Ruby into a completely alien language), but it's also a very nice property.

Unlike Rust where the compiler constantly shouts "fuck you" even though you are trying to do your best to serve their majesty and the rules they dictate, Ruby never gets in your way.

Ruby lets you express what you want to do, in a simple and natural (for your own definition of "natural") way.

Ruby is frequently described as being optimized for developer happiness. And this is definitely true. Ruby lets you write things that work. Quickly. Effortlessly.

This is not a niche language. The number of available modules (`gems`) is impressive, and finding the toolset you need to build an application is rarely an issue.

Sure, Ruby code will not run as fast as some other languages. But in order to get things done, Ruby is second to none. And an application that can actually be used has more value than something that runs faster, but is half-baked.

During my time at OpenDNS (now Cisco), I used Ruby for absolutely everything. I was part of the team crunching data, and we had the luxury to be free to use whatever we wanted to get the job done. So I picked Ruby.

Even the Hadoop jobs were written in Ruby (thanks, JRuby).

Oh sure, all these things would have run way faster if they had been written in Java. But Ruby was the perfect language to quickly try stuff, and incrementally improve it.

The DNS database that eventually became a core company product and something Cisco is now selling like hot cakes is a weekend project I built with Ruby. I just wanted to build this to run some experiments on the data we had.

Would I have done it at all if I had used a less productive language? Probably not. That would have taken more than a weekend, and I probably wouldn't have done it on my spare time.

I rewrote parts of it later in C and Rust to get more speed. But once again, without the productivity and the developper-friendliness of Ruby, the project may have never existed.

Most of my opensource projects were in C or PHP at that time (plus a bunch of languages I flirted with, such as Erlang). But all the closed-source scripts I ran on my laptop and servers were written in Ruby. My personal websites were built with Ruby as well. Because I just wanted these scripts to do what I needed, and not spend much time writing them.

Go

When the first public versions of Rust were announced, I gave it a spin. Just because trying new languages is interesting. All languages suck, but they all bring interesting concepts as well, that can overall make you a better programmer.

Rust was the complete opposite of Ruby from a developer-friendliness perspective. And the first versions of Rust were quite different from today's Rust. I took it as a challenge, and started writing more and more Rust. I also learned Scala, that I had to use with Spark during my time at OVH. Ruby wasn't much in use there. I still had (and have) a decent amount of things written in Ruby running on my servers, but these don't require much maintenance.

So I had less and less opportunities to use Ruby. But I had more and more opportunities to write Rust, even though the language eventually made me a grumpy person rather than a developer having fun as when I used Ruby.

At some point, I wanted to completely rewrite `dnscrypt-proxy`.

I wrote the original version in C, but maintaining that code was painful. Even with great libraries, writing async network code in C always requires a lot of code for little results, besides ensuring that everything will run reliably and securely.

Some features had been planned for a while (such as Anonymized DNS), but I didn't manage to find time to implement them. C is a fantastic system language, but not the most productive one for everything else. Adding even small features would have taken more time that I had to spend on this.

Another downside of using a language such as C is that the community of C developers is dying. Number of pull requests to improve the code or add features to `dnscrypt-proxy` since the project existed: zero. Even though there was a quite large community of users, including corporate users.

I decided to go with… Go.

I had no experience with Go at all. All I knew is that it had that concept of "goroutines", and that looked interesting. Plus, compilation was fast.

That decision was heavily criticized by former `dnscrypt-proxy` users. "Go is bloated! Look at how large the binaries are! Look at the memory usage! And it's probably slow!"

But in retrospect, choosing Go was probably the best decision I've ever made, and I would totally choose it again today. Let me explain why.

Go is more opinionated than Ruby. Unlike languages that encourage complex constructions to show off your knowledge of the language, Go code is always simple and readable.

Go is stable. Backward compatibility is critical. So, in order to support new systems, or improve my applications speed, all I need is recompile with a new version.

The Go compiler is fast. And that proved to be incredibly useful in order to quickly iterate, by testing minor code changes immediately after having written them.

Tools to write Go code work amazingly well. VSCode automatically adds imports (not always a good idea, though, more on that later), always finds the correct type for any expression and autocompletion works (and it feels great! Especially coming from Rust). The documentation is excellent, with an emphasis on examples rather than internal details.

Even though I had no prior Go experience, I managed to quickly get something that was doing exactly what I wanted. The C version of `dnscrypt-proxy` was about 5 years old at that point, but I rewrote something from scratch that did everything it did and much more in about a week. I got a basic proxy implementing the protocol after 15 minutes.

In addition to the Go language and its tools, what made that possible is the vast amount of readily-available modules, as well as its excellent standard library. And, yes, goroutines.

Go turned out to be a super productive language. Something I enjoy writing because I can see my application evolve the way I want it to without having to write a lot of code, or thinking too much about how to make a borrow-checker happy before adding a single line.

Another amazing thing about Go is its portability. Sure, many languages are also highly portable, but Go libraries are all designed with portability in mind, emulating what's missing if required, but offering a unified interface for everything.

I mainly use MacOS to write my code, but with Go, I can be pretty confident that the same code will run exactly the same way on any supported platform.

And cross-compilation using Go is way easier than any compiler I've ever seen. Each release of `dnscrypt-proxy` is now automatically packaged for 23 targets, because this is absolutely trivial to do, and once again, I can be confident that it will run exactly the same as on my development platform.

Go has pointers. But the code produced by Go includes bounds checking everywhere. Unlike C, if a pointer is not used correctly, the process will safely crash, with a comprehensive stack trace.

Speaking of stack traces, and debugging in general, Go is absolutely great. Stack traces are short, readable, and contain everything you need to understand the root cause of a crash. Debugging code with Visual Studio code and `delve` is great. It just works.

The ecosystem is big. Whenever I need something, there's a module for it. The community is big. Whenever I'm stuck, there's an answer somewhere.

I found in Go what I originally found in Ruby. A language that lets me express what I want to do in a natural way. My applications improve quickly, and programming makes me happy again.

"But Go has a GC!"

Yeah, so what? GC is what makes the language not a PITA to use. But, just like in Java, if I need speed, I can preallocate and reuse memory to avoid hitting the GC.

I can write whatever I want to write in a simple and natural way first, and then take the time to optimize it to avoid GC pauses later if needed.

As a result, Go is not slow. See the FastHTTP package for example.

"But it doesn't prevent data races!"

Rust forces me to write in a way that doesn't feel natural. I found myself spending too much time making the borrow checker happy without this solving any actual issues that would have occurred at runtime. I sometimes appreciate the rigidity of a language forcing me to design my code in a certain way to prevent

issues. But I also appreciate a language that lets me implement the mental model I have without getting in the way, especially when great debugging tools are available.

Go's productivity allows me to focus more on the algorithms than their implementation. Overall, this is a win. `dnscrypt-proxy v2` has been reported to be way faster than the first version in C.

But more importantly, it has tons of features. That I would have never ever implemented if I hadn't rewritten it in Go.

For me, Go has become the new Ruby. A language I use to get things done, and enjoy programming again.