Fortune Apps

Adrian Kosmaczewski

2022-05-20

As part of my work in VSHN, I lately prepared a set of demo applications ready to be containerized and deployed in our new product APPUiO Cloud.

The applications are available in GitLab. They have all the exact same set of features (not a lot, mind you), while being written in ~~15~~ 16 different programming languages.

1. C using the GNU libmicrohttpd system
2. C# using the standard ASP.NET framework
3. C++ thanks to the Drogon web framework
4. Dart using the Conduit framework
5. Elixir using the Phoenix Framework (mentioned in a previous article)
6. F# using Giraffe
7. Go based on Gin
8. Java using Red Hat's Quarkus
9. Kotlin based on Ktor
10. PHP with the Slim Framework
11. Python using Flask (mentioned in a previous article)
12. Ruby with Sinatra
13. Rust using Actix and Askama (metioned in a previous article)
14. Scala based on Scalatra
15. Swift using the Vapor framework
16. TypeScript based on Express

Each and every one of these applications are fully containerized, and ready to run. They all offer exactly the same features:

- A simple API returning a `fortune` message, and a random number, on port 8080.
- The API responds JSON when called with `Accept: application/javascript`, plain text when called with `Accept: text/plain`, and just HTML in all other cases.
- It uses server-side rendering, because it's hype again, apparently.

I could not help myself getting some statistics out of these projects, and making a quick and dirty classification of these programming languages and frameworks.

I gathered the following data points:

- The energy ranking provided in this page;
- The memory consumption as shown in Docker Desktop;
- The lines of code of the source code of the app, counted by cloc;
- Build times and final size of the Docker container as shown in GitLab.

I put all the data in a LibreOffice spreadsheet, multiplied the 5 values above all together into a new `score` column, and ordered everything in ascending order. The build time was interpreted as a timestamp in December 1899 or something like that.

And the results are here; the lower the score, the "better".

Language	Energy Rank	Mem (MB)	LOC	Build (h:m:s)	Size (MiB)	Score
Rust	1.03	1.00	66	00:11:20	7.33	3.92
C	1	4.11	102	00:02:30	5.95	4.33
Go	3.23	4.77	66	00:02:00	13.22	18.67
Dart	3.83	8.40	76	00:02:35	9.42	41.33
C++	1.34	3.04	79	00:16:06	17.52	62.93
Java	1.98	6.80	133	00:15:09	25.96	488.86
C#	3.14	33.47	88	00:01:48	46.67	539.53
PHP	29.3	24.98	37	00:01:36	26.1	785.34
TypeScript	21.5	14.67	46	00:02:07	43.75	933.03
Python	75.88	19.77	30	00:01:24	22.54	986.22
Ruby	69.91	28.17	20	00:01:22	29.93	1118.83
F#	4.13	31.48	107	00:02:10	55.63	1164.41
Scala	1.98	96.61	70	00:03:08	78.16	2277.27
Swift	2.79	6.15	110	00:15:45	153.86	3173.68
Kotlin	1.98	308.80	80	00:04:54	69.57	11579.45
Elixir	42.23	85.03	414	00:03:02	62.21	194810.09

Taking everything into account, I'd consider using **Go, C#, or TypeScript** (in that order) for such an API anytime. With each of these three languages, the developer experience is unparalleled, the tooling is excellent, compilation times are super short, the ecosystems are huge and vibrant, and the final results are quite good (by that I mean low energy rank, low memory requirements, low SLOC, and fast build times.)

Why Go, C#, or TypeScript, and not the other programming languages?

- Rust, Java, C++, and Swift build times are too long (any relationship to strong type checking is just a coincidence). But yeah, nobody beats Rust's low memory requirements. To be honest, if Rust's compilation times go down in the future (and I bet they will), it would become the one and only language to consider in this whole list.
- C is… C. Even though the compilation is really quick.

- PHP is… PHP. But the scores of the final product aren't bad at all, don't get me wrong. I kinda like PHP, but wouldn't use it in this context.
- The framework chosen for F# (Giraffe) is a bit behind in terms of documentation. I should have just used ASP.NET here, in which case F# would go up in my personal opinion ranking.
- Elixir, Scala, and Kotlin use too much memory. The interesting thing about Scala and Kotlin is that they are running in a container image with the full Java runtime in it, while the Quarkus container (built with the Java programming language) was just running a self-contained binary, with dramatically lower requirements. No Java runtime, low memory and fast startup; that's precisely Quarkus' selling point.
- Elixir, Python, and Ruby are too high in the energy efficiency index (even though Ruby has the shortest SLOC of all)

Some other observations you might find interesting:

- All container images are based on Alpine except for C and C++ (based on `busybox:glibc`), Java (based on `quarkus-distroless-image:1.0`), Swift (based on Ubuntu 20.04, because Swift does not yet work with musl), and Elixir (based on Debian).
  - The idea was, of course, to have the smallest possible container image with the least possible effort and keeping some readability in the `Dockerfile`s; of course you could make these images smaller, but at the expense of debuggability or understandability. Feel free to use those `Dockerfile`s if you find them useful.
- Python, Typescript, Rust, and PHP use exactly the same HTML templating syntax (and Go's is almost identical).
- Python, Go, Scala, F#, Ruby, and PHP use exactly the same `sprintf()` format, because POSIX and whatnot.
- Some of these apps were a true PITA to build: in particular, I'll mention Swift. The biggest issue was the low quality and quantity of the Vapor documentation; another issue is the slow compile-test cycles (this was done in Linux, not on a Mac, maybe Xcode can help there, but on Visual Studio Code, no help at all).
  - By the way, the `Dockerfile` of the Swift project is the one generated by the Vapor bootstrap command. Haven't touched it further.
  - Hopefully Swift will be compatible with musl one day (some adventurous people are trying to do it on behalf of Apple), to enable Alpine-based images.
- The easiest ones to build: Python, Ruby, and PHP. Scripting languages are such an easy thing to use, seriously. Although Go, C#, and TypeScript, were also very easy to work with, and with the added benefit of a nice type system on top.
- All images were built for the x86-64 architecture.
- The average SLOC is 96.5 (standard deviation 93.3), and the median 79 (corresponds to the C++ project).
  - The smallest SLOC counts are those of scripting languages: Ruby,

Python, PHP, and TypeScript, in that order. They also have very high energy rankings. What you get on one side, you lose on the other.

- The Elixir project contains too much boilerplate code I haven't removed, which contributes heavily to the SLOC. After all, the Phoenix Framework is much more similar in spirit and functionality to Ruby on Rails (or Django) than Sinatra (or Flask).

Finally, I used the code of all of these projects to automatically generate the documentation shown in the Getting Started page of APPUiO Cloud, thanks to the magic of Asciidoctor.

Update, 2022-06-17:: I've added a version using Dart. The corresponding data appears in the table above. It gets a very promising 4th place in the table, thanks to a very fast build process, a small final container size, and very low runtime memory requirements. Unfortunately the ecosystem is not very strong around Dart, but otherwise this looks like an excellent choice for a web API.