Programming languages

Summary

Issue

We need to choose programming languages for our software. We have two major needs: a front-end programming language suitable for web applications, and a back-end programming language suitable for server applications.

Decision

We are choosing TypeScript for the front-end.

We are choosing Rust for the back-end.

Status

Decided. We are open to new alternatives as they arise.

Details

Assumptions

The front-end applications are typical:

- Typical users and interactions
- Typical browsers and systems
- Typical developments and deployments

The front-end applications is likely to evolve quickly:

- We want to ensure fast easy developments, deployments, iterations, etc.
- We value provability, such as type safety, and we are fine doing a bit more work to achieve it.
- We do not need legacy compatibility.

The back-end applications are higher-than-typical:

- Higher-than-typical goals for quality, especially provability, reliability, security, etc.
- Higher-than-typical goals for near-real-time, i.e. we do not want pauses due to virtual machine garbage collection.
- Higher-than-typical goals for functional programming, especially for parallelization, multi-core processing, and memory safety.

We accept lower compile-time speeds in favor of compile-time safety and runtime speeds.

Constraints

We have a strong constraint on languages that are usuable with major cloud provider services for functions, such as Amazon Lambda.

Positions

We considered these langauges:

- C
- C++
- Clojure
- Elixir
- Erlang
- Elm
- Flow
- Go
- Haskell
- Java
- JavaScript
- Kotlin
- Python
- Ruby
- Rust
- TypeScript

Argument

Summary per language:

- C: rejected because of low safety; Rust can do nearly everything better.
- C++: rejected because it's a mess; Rust can do nearly everything better.
- Clojure: excellent modeling; best Lisp approximation; great runtime on the JVM.
- Elixir: excellent runtime including deployability and concurrency; excellent developer experience; relatively small ecosystem.

- Erlang: excellent runtime including deployability and concurrency; challenging developer experience; relatively small ecosystem.
- Elm: looks very promising; IBM is publishing major case studies with good resutls; smaller ecosystem.
- Flow: interesting improvement over JavaScript; however; developers are moving away from it.
- Go: excellent developer experience; excellent concurrency; but a track record of bad decisions that cripple the language.
- Haskell: best functional language; smaller developer community; hasn't achieved enough published production successes.
- Java: excellent runtime; excellent ecosystem; sub-par developer experience.
- JavaScript: most popular language ever; most widespread ecosystem.
- Kotlin: fixes so much of Java; excelent backing by JetBrains; good published cases of porting from Java to Kotlin.
- Python: most popular language for systems administration; great analytics tooling; good web frameworks; but abandonded by Google in favor of Go.
- Ruby: best developer experience ever; best web frameworks; nicest community; but very slow; somewhat hard to package.
- Rust: best new language; zero-abstraction emphasis; concurrency emphasis; however relatively small ecosystem; and has deliberate limits on some kinds of compiler accelerations e.g. direct memory access needs to be explicitly unsafe.
- TypeScript: adds types to JavaScript; great transpiler; growing developer emphasis on porting from JavaScript to TypeScript; strong backing from Microsoft.

We decided that VMs have a set of tradeoffs that we do not need right now, such as additional complexity that provides runtime capabilities.

We believe that our core decision is driven by two cross-cutting concerns:

- For fastest runtime speed and tightest system access, we would choose JavaScript and C.
- For close-to-fastest runtime speed and close-to-tightest system access, we choose TypeScript and Rust.

Honorable mentions go to the VM languages and web frameworks that we would choose if we wanted a VM lanauge:

- Clojure and Luminus
- Java and Spring
- Elixir and Phoenix

Implications

Front-end developers will need to learn TypeScript. This is likely an easy learning curve if the developer's primary experience is using JavaScript.

Back-end developers will need to learn Rust. This is likely a moderate learning curve if the developer's primary experience is using C/C++, and a hard learning curve if the developer's primary experience is using Java, Python, Ruby, or similar memory-managed languages.

TypeScript and Rust are both relatively new. This means that many tools do not yet have documentation for these languages. For example, the devops pipeline will need to be set up for these languages, and so far, none of the devops tools that we are evaluating have default examples for these langauges.

Compile times for TypeScript and Rust are quite slow. Some of this may be due to the newness of the languages. We may want to look at how to mitigate slow compile times, such as by compile-on-demand, compile-concurrency, etc.

IDE support for these languages is not yet ubiquitous and not yet first-class. For example, JetBrains sells the PyCharm IDE for first-class support for Python, but does not sell and IDE with first-class support for Rust; instead, JetBrains can use a Rust plug-in that provides perhaps 80% of Rust language support vis a vis Python language support.

Related

Related decisions

We will aim toward ecosystem choices that align with these langauges.

For example, we want to choose an IDE that has good capabilties for these languages.

For example, for our front-end web framework, we are more-likley to decide on a framework that tends to aim toward TypeScript (e.g. Vue) than a framework that tends to aim toward plain JavaScript (e.g. React).

Related requirements

Our entire toolchain must support these languages.

Related artifacts

We expect we may export some secrets to environment variables.

Related principles

Measure twice, build once. We are prioritizing some safety over some speed.

Runtime is more valuable than compile time. We are prioritizing customer usage over developer usage.

Notes

Any notes here.