Choosing a (Language) Stack

Nick Gordon Follow
Sep 11, 2018 · 13 min read

Kotlin vs. Go vs. Ruby

This post and the experiment was a collaborative effort by Michael Farinacci (Load Testing), Nick Gordon (Ruby), Iain McGinniss (Kotlin), and Kevin Steuer (Go).

Introduction

Our team has been tasked with building the next generation of Identity infrastructure for WeWork. With the anticipated rate at which WeWork and its sibling businesses are growing, it is important that this new system be efficient, globally distributed, scalable, and permit reasonable evolution as requirements change over the coming years.

Based on these challenges, the team felt it was worth investing the time to compare the available languages and software stacks for building microservices. We had three candidate languages: Go, Kotlin, and Ruby. The comparison was made by building the same realistic component of an Identity system in each. We chose a token mint service, which produces bearer tokens like refresh and access tokens in an OAuth2 based infrastructure. Each language has some precedent in other systems built at WeWork, and each implementation was written by a developer with experience in that language.

Upon completion of these implementations, they were compared across the following qualitative and quantitative metrics by the team:

- Qualitative comparison of the readability of the code.
- Qualitative comparison of how straightforward it was to test each implementation, at a unit and integration test level.

- Qualitative comparison of the “boilerplate” necessary in each implementation, versus the code that represents the essence of the service.
- Qualitative comparison of the tools available for each stack, such as IDEs, static analysis, debuggers, profilers, etc.
- Quantitative comparison of the performance of each implementation, such as round trip time percentiles and host resource utilization.

Ultimately, an experiment like this cannot be wholly scientific—software engineering is influenced by sociological factors as much as by science and mathematics. Our goal was primarily to pick a good all-rounder language and tool set, that the team will be comfortable working with over the next five years.

The Token Mint Service

The service selected for this exercise was a “token mint”, which produces three types of bearer tokens: refresh tokens, access tokens, and “action” tokens. Isolating token production and verification into its own service can help to isolate the use of cryptographic keys, that are critical to system-wide security.

The refresh and access tokens produced by this service have the usual OAuth2 semantics: refresh tokens do not expire but can be revoked and are used to produce access tokens that expire after 18 hours. “Action” tokens are a non-standard addition; access tokens are exchanged for action tokens in the context of a single external API call. We would perform this exchange using a request interceptor at the API gateway (we use Kong), in order to help to decrease the risks associated with accidental or malicious token logging or exfiltration in misbehaving downstream services.

The produced token values, from the perspective of any other system or client, are just url-safe ASCII strings. They do, however, have internal structure. They are Base64 encodings of a binary protocol buffer, that contains token metadata and a digital signature. The signature algorithm used in the experiment is ECDSA, over the NIST P-256 curve.

The service offers five operations to other authorized components of the Identity system, over gRPC:

- Mint a refresh token for a specified user
- Revoke a refresh token
- Mint an access token, given a non-revoked refresh token
- Mint an action token, given a non-expired access token
- Get info on a supplied token

The service is simple enough to be implemented in a few weeks by a single engineer, while being representative of the real systems we must build:

- The service must interact with a database to store information on the tokens it generates, and to remember which refresh tokens have been revoked.
- The service must handle multiple concurrent requests, and be able to scale and distribute to handle thousands of requests per second.
- As a security sensitive service, inputs and outputs must be carefully validated, and careful use of cryptographic primitives is necessary.

A system of this nature is primarily I/O bound and performs database writes and queries over a network. The most compute-intensive operations are the generation and validation of digital signatures. Production and validation of token metadata and protocol buffers is computationally trivial in comparison.

Load testing

A load test client was implemented in Go, using gRPC client stubs generated from the service definition. The load tester simulated a population of users interacting with the system, given a reasonable approximation of expected usage patterns.

For one user, we may expect (on average) one refresh token request per six months, one access token request per day, and one action token request per hour. Combining these assumptions with a simulated population size p produces an estimation of the average load on the system: $l(p) = p \times 2.894 \times 10^{-4}$ requests per second, with 95% of those requests being action token requests. Given a population size of 100k users, the load tester would generate roughly 29 token requests per second. By adjusting the simulated population size, and/or running multiple load test processes in parallel (on different machines, if necessary), the load on the service was adjusted.

The load testing client recorded the round trip time for every request made in microseconds, and any errors that occur. This information was then used to produce request scatter plots, histograms, and other statistics. We could tell when a service starts to reach its scaling limits when its average latency and/or 90th percentile starts to climb noticeably—this typically indicated some form of resource starvation.

Load Test Results

The database started to throttle throughput at a population size of around 1.5 million, we compared the performance of the implementations at 1.0 million. At this population size, the load testing client attempted to generate an average of 289.4 requests per second. The most interesting stats were:

Search this file…

Implementation	CPU usage	RPS	Median (μs)	95th (μs)	99th (μs)
Kotlin	50%	287	4916	7484	11140
Go	31%	296	4484	6352	9246
Ruby	41%	274 (ex. errs)	4720	6713	9368

load_test_results.csv hosted with ❤ by **GitHub** view raw

Load Testing Performance Numbers (Population of 1.0 million)

Here, we can see each implementation offers very similar median response times, within fractions of a millisecond. However, the Kotlin implementation starts to exhibit measurably worse 95th and 99th percentile latencies; we speculate that this may be due to the context switches involved in passing data between the network, coroutine execution and JDBC query thread pools that the implementation used. Such context switching was unnecessary in both the Go and Ruby implementations. It was also clear that Go required the least compute time of the three implementations to handle the supplied load. Ruby's CPU usage stats may have been slightly misleading, as the errors (~3% of requests) produced had minimal impact on CPU usage: requests were rejected prior to any real processing.

Load Test Conclusions

Despite the preconceptions we may have had, it is clear that each stack was perfectly capable of implementing a performance sensitive service. Each implementation, with a single container on modest hardware, was able to handle a population size that we do not expect to see at WeWork until 2021, and could likely handle significantly more with some optimization of our database usage patterns.

Ruby's use of native code for the most compute intensive parts of the implementation mean that its interpreted nature is barely noticeable; it is only the quality of the Ruby gRPC implementation that let it down. The Go implementation appears to be the most efficient of the three implementations, and had the most predictable performance characteristics as we increased load on the service.

All three implementations could likely be optimized further—these load test results were produced with no specific attempt to profile or improve the performance of the implementations. As the implementations stand, Go has a slight edge. The JVM has the best tools for profiling and the most configuration options to squeeze performance out of a running system, but such optimization is brittle; it is better to invest that time in performance optimization at the architectural level.

Overall, we make the banal conclusion that performance has much more to do with the choice of approach than the choice of language. Good performance can be achieved in almost any stack, by skilled engineers who understand its nuances.

And the winner was…

Go, by the team's majority vote. Kotlin came second, Ruby third. The team felt that Go was the better technology choice for the following reasons:

- A simpler language naturally constrains programmers to simpler code, which will make it easier to review and maintain on a day-to-day basis. We need strong recommendations on tool choice and architecture to ensure this extends to higher-level concerns—simpler code, in aggregate, can be harder to understand if care is not taken in how it is composed.
- The development lifecycle is typically simpler with Go—there is little difference between running a service directly on a dev machine, to running it in a container, to running clustered instances of the service. Containers produced for Go microservices are very compact, as Go binaries have essentially no external requirements aside from POSIX OS interfaces.
- Go provided marginally better performance than the other languages, with minimal fuss. Its built-in asynchronous processing mechanisms and associated libraries appear to make it better suited to the types of microservices we are likely to write in the Core Platform group.

This choice comes with the strong caveat that we must provide rules or recommendations on:

- Dependency management. We are likely to adopt go modules for this purpose, once it is ready.
- Error propagation and handling—this is a notoriously weak part of the language, for which we must define a workable approach.
- Channel and goroutine usage—beginners can tie themselves in knots with this. However, it is no worse than async/await in EcmaScript, or ReactiveX patterns; this is rapidly becoming a normal part of software engineering practice.

Implementation Details

The following sections are in-depth discussions of each implementation, with the various pros and cons of each specifically discussed.

Evaluating Kotlin

Kotlin is a statically typed, object-functional language designed by JetBrains, first released in 2011. It is primarily compiled to JVM bytecode, but has experimental support for compiling native binaries (via LLVM), and transpiling to Javascript.

We regard Kotlin as a more flexible, less verbose alternative to Java, and a simpler alternative to Scala. It has rapidly grown in popularity in the Android ecosystem, where it is now supported by Google as a first-class citizen for Android apps and libraries.

Implementation notes

Three key decisions were made in the Kotlin implementation, that had a strong influence on the structure of the resultant code and likely on performance:

- Frameworks were avoided in the code, with the exception of the framework-like elements introduced by gRPC-java itself. The service is simple enough that we felt frameworks such as Spring Boot, Lagom or

Vert.x would do little but obfuscate the implementation, particularly for those not familiar with Kotlin or the JVM. As a result, the initialization logic and interaction with the database was more explicit than what one may have found with the use of a framework.

- The Arrow functional-programming library was used extensively to facilitate a mostly-functional style. In particular the Either type was used pervasively instead of thrown exceptions for errors. This gave the code a Go-like feel, with explicit error checking on method calls, though Arrow's monad comprehensions or flatMap chaining were used to make this less disruptive to readability than the equivalent imperative Go code.
- Coroutines were used, in combination with async/await, in an attempt to simplify the concurrent and asynchronous parts of the code.

Strong Points

Kotlin is a good all-rounder language: linguistically it matches Ruby's expressiveness, while retaining strong static type safety. Its linguistic features allows for more compact, expressive code than Java, without being overly opinionated on the OOP / FP dichotomy. This linguistic flexibility is particularly well suited to building domain-specific abstractions to help in solving problems. This is also a core strength of Ruby, and arguably a core weakness of Go.

Extension methods are a great feature, and helped with the readability of the code. A "fluent" style of programming through method chaining can be easily added to existing types through the use of extension methods, which we put to good use in implementing builders and validators for protocol buffer messages.

Straightforward interop with JVM libraries, tools and frameworks is also a core strength: Java programmers can get started in Kotlin in minutes, and even if they do not adapt their programming style to include functional patterns, they can see meaningful productivity gains. It is also not surprising that Kotlin has an excellent IDE, given its origin at JetBrains.

Pain Points

While a flexible, pragmatic approach to functional programming in Kotlin is a distinct strength, it can also be viewed as a weakness: it is easy to get carried away with the functional style, and end up with difficult-to-read code. This happened in the implementation where we went too deep with usage of the Arrow library—while equivalent imperative code may be longer, it can be easier to understand and less nuanced.

As a relatively new language that gives the programmer a wide variety of linguistic tools, there are no well-established best practices for using the language. As a result, this *will* lead to wildly different coding styles and approaches to problems across projects, teams and individuals. Go and Ruby, by comparison, have more established idioms.

Some awkward issues were found in using Kotlin coroutines, which are still an experimental part of the language. In particular, composing coroutines with Arrow's Either type monad comprehensions is difficult. Arrow's EitherT monad transformer helps but is somewhat magical to a layman.

Some of the core Java APIs are starting to show their age, and do not fit well with contemporary asynchronous programming. JDBC is a prime example: operations like acquiring a database connection and dispatching queries are blocking operations. Until the asynchronous successor to JDBC is available, JDBC calls must be made via a separate thread pool to avoid blocking coroutine threads.

Evaluating Go

Go is a statically typed, imperative language that was first released by Google in 2009. It was motivated by the need to solve large-scale software engineering problems at Google, and is typically compiled to statically-linked native binaries.

We regard Go as a contemporary alternative to C/C++, that preserves linguistic simplicity, while providing built-in essentials like garbage collection and asynchronous programming primitives. Go is being adopted by many engineering organizations for backend systems, including Uber, Slack, Dropbox, and Twitch.

Implementation notes

We used the go grpc framework, leveraging go-grpc-middleware, handling our common infrastructure patterns for handling panics, opentracing, and request attribute validation. This resulted in a clean common server pattern that we can reuse across our microservices.

For our libraries we adopted the functional options pattern with smart defaults, inspired by Dave Cheney's blog post and gRPC.

Strong Points

Strong points are covered under why we chose Go, with a few additional reasons:

- Concurrency principles are built in. Channels and goroutines enables concurrent applications to be written with great readability.
- Auto formatting. We don't have to discuss tabs vs spaces :)
- godoc. Documentation is built in as a standard to the go tooling

Implementation Pain Points

For the implementation the biggest tooling pain point encountered was around the database unit tests using go-sqlmock. Initially the SQL statements were written for the MYSQL flavor vs Postgres and wasn't caught until the integration tests ran. Another approach may be to use an in-memory db database that speaks the postgres protocol.

Errors in Go are arguably too simple and error checks seem to be tedious and represent a large portion of the code. This results in teams coming up with their own error types or strategies.

Fragmentation of dependency management tools causes confusion on which one to adopt. Go modules was recently shipped in Go 1.11 so this may be fully addressed.

Evaluating Ruby

Ruby is a dynamically typed, object oriented, interpreted scripting language. It was first released in 1995 by Yukihiro "Matz" Matsumoto, who remains heavily involved in the design of the language to this day. Ruby's interpreter is written in a mixture of C and Ruby, with the ability for packages (called "gems") to include native extensions, such as C, C++, or Rust.

We regard Ruby as one of the more flexible dynamically typed languages and the company has a large amount of experience with it. The majority of WeWork's applications (as of June 2018) are written in Ruby, primarily using Ruby on Rails.

Implementation notes

No framework was used in the Ruby implementation, but the code was organized in a fashion that would be familiar to Ruby on Rails developers (model, controller, services, etc.). Due to the simplicity of the data models, a simple wrapper around the pg gem was written, rather than using activerecord or other DAO/ORM libraries.

We used gruf, written by BigCommerce, to manage gRPC requests. It abstracts out the majority of the boilerplate-level concerns with managing GRPC servers while still exposing the underlying configuration if desired.

Multithreading/Horizontal Scaling

Asynchronicity is handled purely at the server level—each request operates in its own thread from the Controller down to the Database calls. The interpreter has a global lock, and therefore more complex threading is generally not idiomatic in application-level Ruby code. This may cause the alternate implementations to scale horizontally more easily.

The Docker images are noticeably larger (a comparable deploy image using alpine is still 2.5x the size of the Kotlin or Go containers), but this is not a huge difference in most deployments since only the initial pull to the orchestration will be dealing with the network time.

Ruby gRPC server (using the official library) does not queue requests (since the merge of this pr), which means that under heavy load it will reject many incoming requests.

Strong Points

As part of the test suite, integration tests were written which were easy to package as an acceptance test suite for other implementations of the suite. RSpec is very well supported and easily extensible, while producing readable failures for non-ruby devs as well.

While the database logic in this project was straightforward enough that the author did not feel it necessary to do anything more complicated that the base Ruby postgres gem, more complex database projects could switch to using ActiveRecord and take advantage of the ORM that makes Rails so powerful.

While OpenTracing was not added into the project at the time of this writing, adding in NewRelic tracing and monitoring required adding a simple Gruf interceptor. Additional instrumentation was also added to fine-tune and required minimal changes to the code.

Pain Points

Much of the message contents for this project were around Bytes and Cryptography.

Dealing with UUIDs as bytes is not difficult to do, but it is not a first class citizen. This required writing a few helper methods, which were simple to write but were an additional overhead in the context of a project like this.

Cryptography in Ruby is generally done through system libraries (Ruby NaCl or OpenSSL), but this project in particular specified Tink. Since Tink is not implemented in Ruby (only in C++, Java, and Go at the time of this posting), multiple imports and handlers were needed to match the other implementations.

A major pain point in the syntax of the Ruby generated code has been resolved as of the writing of this analysis: *ruby_package* can now be used to declare namespaces dynamically, making the Ruby code a bit more idiomatic (release and gRPC pr).

Interested in joining our team? WeWork is hiring software engineers in Tel-Aviv, New York and San Francisco. View openings