GO @ EXOSCALE

VINCENT BERNAT — EXOSCALE

EVALAIS.CH — MARTIGNY — 2018-03-27

THE GOOD PARTS ❤️

EASY TO LEARN

- Simplicity
- Not many concepts to grok
- Fluency in a few hours
- Code usually readable (a bit verbose)
- Good practices are fairly understood

Recommended reading:

- A Tour of Go

CONCURRENCY

- **Goroutines**: light-weight threads
- **Channels**: share data between goroutines
- Classic primitives are available if needed (mutex locks…)
- Beware of goroutine leaking

Recommended reading:

- Go channels are bad and you should feel bad
- Death of goroutines under control

SAFETY

- Memory-safe
- Garbage-collected
- Static typing
- Test culture

PERFORMANCE

👌

TOOLING

- `gofmt` will format your code
- `go test` has a race detector
- `go build` supports cross-compilation (build a Linux executable for your Raspberry Pi)

GREAT ECOSYSTEM

- Need a Zookeeper client? go-zookeeper
- Need a PostgreSQL client? pq
- Need a SSH server? ssh
- Need a BGP daemon? gobgp
- Ability to interface with C easily

THE “MEEEH” PARTS 🙄

STANDARD LIB

- Some parts are not great:
  - logging
  - command-line parsing
  - testing
- Some parts are great, notably HTTP

NOT REALLY A SYSTEM LANGUAGE

- Standard library abstraction to support Plan 9
- Breaking abstraction is sometimes difficult
- Runtime can get in the way: until recently, namespaces were mostly unusable

DEBUGGING

- No good story so far for debugging
- Most C tools like `gdb` and `perf` work with Go

THE BAD PARTS 🧟

GOPATH

- Go enforces the way you organize your files
- Your code is mixed with your dependencies
- Some people like it, some hate it
- Workaround with some `Makefile`
- Will go away soon (part of `vgo` plan)

NO GENERICS

- Difficult to write generic algorithm without them
- Due to compatibility promise, they'll never be implemented
- Go builtins are using generics (`append`, `make`)
- Instead, people use interfaces (no more type safety at compile time)
- Also see: sort.Slice

NO VERSIONING CULTURE

- Strong culture of “backward compatibility”
- But some projects don't care about that much
- Also, no way to know if the version you are using is stable (in the middle of a refactor?) or very different from the version of last month (major rewrite?)
- But versioning is coming (part of `vgo` plan)

DEPENDENCY MANAGEMENT

- Python: `pip`. Ruby: `bundle`. Java: `mvn`
- During a long time, for Go, only `go get`
- Vendoring was enabled in Go 1.6 (dependencies in `vendor/`)
- Many different tools were proposed by the community (`godep`, `glide`, `gb`)
- In 2016, `dep` was started as the to-be official package manager. Work like Ruby's `bundle` (so good)
- In 2018, the whole experiment is replaced by the `vgo` plan

GO @ EXOSCALE

CLOJURE SHOP

- LISP on top of the **JVM**
- Great interoperability with Java
- Immutability (great for concurrency)
- Most of our in-house products are developed with Clojure

GO?

- JVM is memory and CPU-hungry
- C is error-prone (memory safety) and ecosystem is of inequal quality
- Python may be too slow
- Haskell is difficult for newcomers
- Go is the current best language to develop **system-oriented components**

EXAMPLE: JURA

- Network orchestration
- Cloud orchestrator provides network info for each VM to JURA
- JURA locally configures the network on each hypervisor
- Small codebase: 20k+ lines of code

COMPONENTS

- Build:
  - Makefile for compilation without a `GOPATH`
  - `dep` for vendoring and dependency management
- Reporting:
  - Structured logging: inconshreveable/log15.v2
  - Error handling: pkg/errors
  - Error reporting: raven-go
  - Metrics: rcrowley/go-metrics + go-collectd

COMPONENTS

- CLI: urfave/cli.v1
- Retry: cenkalti/backoff
- Goroutine management: tomb.v2
- Dependency injection: facebookgo/inject + facebookgo/startstop

See also:

- go-kit

QUESTIONS?