Go at Skroutz

Agis Anastasopoulos

Tech Lead, SRE team at Skroutz

agis.io @agisanast github.com/agis

Table of Contents

Who we are

- Leading online marketplace in Greece (est. 2005)
- Operate in 2 countries
- 5M products from 3200 shops
- 1M visitors/day
- 210 employees (85 engineers)

Our users

1. People searching for things to buy
2. People actually buying things
3. Shops

What we do

- Ruby on Rails monolith
  - 70k commits, 120 authors, 10 years of history
- Home-grown infrastructure
  - More at “Skroutz Infrastructure at a Glance”

Challenges

As we grew, some parts did not scale well.

We needed a better platform.

Challenges

What’s the problem with Ruby?

1. Concurrency
2. Performance
3. Standard library

Introducing Go: Why Go?

1. Simple and small language
2. Concurrency
3. Standard library
4. Performance
5. Tooling
6. Go 1 compatibility promise

Introducing Go: How?

1. Pick a good candidate problem:
    a. ...well-defined
    b. ...purely technical
    c. Can be attacked with a single-responsibility system
2. Defer “big” decisions
3. Prototype first
4. Keep it simple

Introducing Go

Current state:

- Using Go in production for the last 2 years
- Five Go projects so far

Project showcase

scratchd

Project #1

scratchd

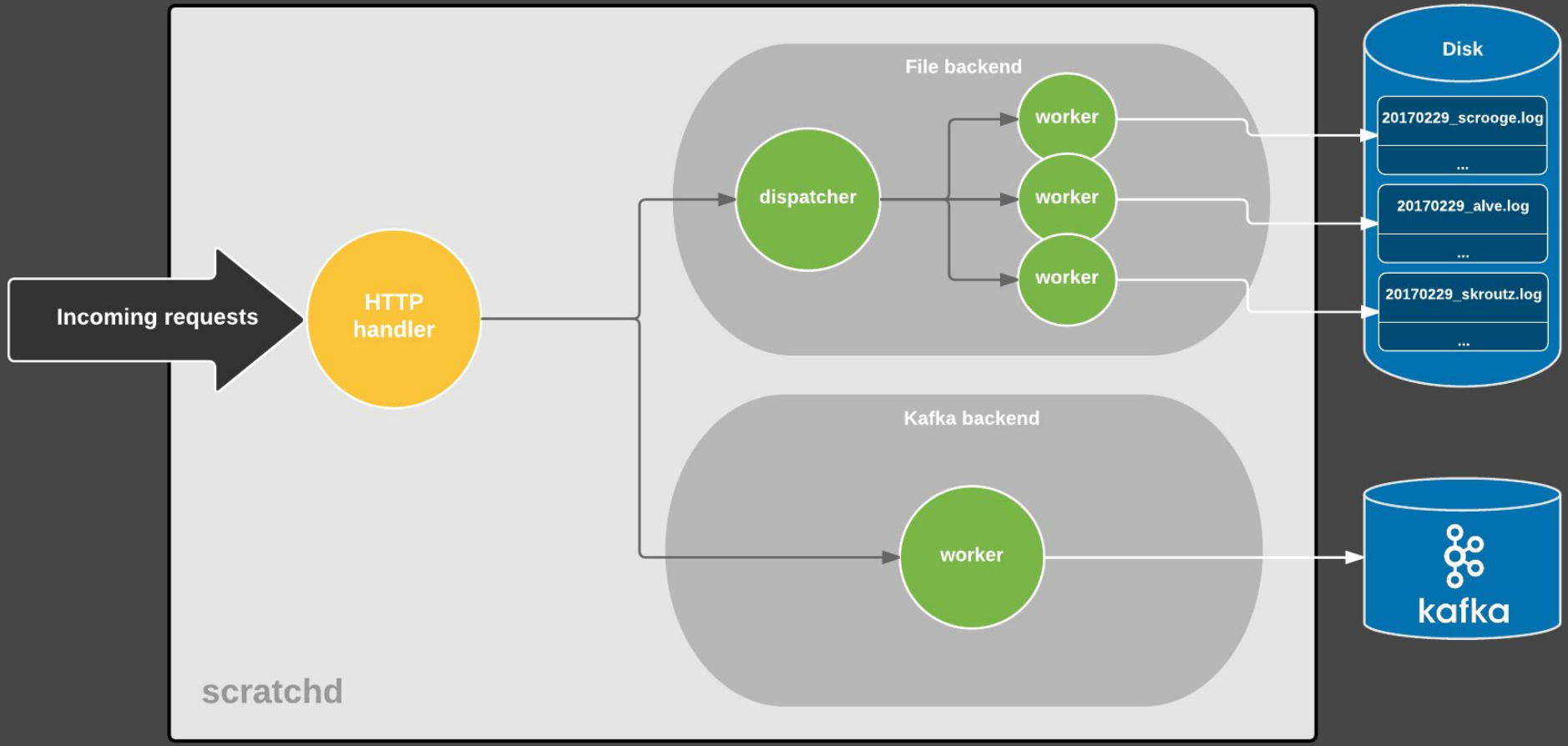
Background:

- The monolith records web analytics and persists them
- Analytics components impact product SLA (bad!)
- Good candidate problem
  - ...well-defined
  - Single responsibility
  - Purely technical

scratchd

At a glance:

- A server that generates and persists web analytics
- Simple, small system with a single responsibility
- Multi-tenant
- Supports multiple persistence layers


Incoming requests
HTTP handler
dispatcher
File backend
worker
worker
worker
Disk
20170229_scrooge.log
...
20170229_alve.log
...
20170229_skroutz.log
...
Kafka backend
worker
kafka
scratchd

scratchd

Lessons learned:

- We verified our assumptions about Go
- Go’s simplicity is a key strength
- Single-responsibility goroutines simplify the overall design

More at “Rewriting our web analytics tracking infrastructure in Go”

rafka

Project #2

rafka

Background:

We want to use Kafka from our Ruby apps.

At a glance:

- Kafka Proxy server with simple semantics
- Abstracts away low-level Kafka details
- Easy to implement drivers

rafka

Lessons learned:

- Allocating a lot of short-lived memory fast can be problematic
  - In detail: https://github.com/skroutz/rafka/issues/39
  - ~~Solution maybe in 1.13: https://github.com/golang/go/issues/16930~~
  - Fixed in 1.12! *"The Go runtime now releases memory back to the operating system more aggressively, particularly in response to large allocations that can't reuse existing heap space."*

rafka

Available at:

- Server: https://github.com/skroutz/rafka
- Ruby Client: https://github.com/skroutz/rafka-rb/
- More info at “Introducing Kafka to a Rails application”

downloader

Project #3

downloader

Background:

- The monolith downloaded product images from merchants
- …based on a *fixed* amount of workers
- No *fine-grained concurrency control*
- Good candidate problem
    - Well-defined
    - Purely technical

downloader

At a glance:

- A service responsible for *downloading files from the internet*
- Rate-limiting capabilities
- MIME type sanity checks

downloader

Implementation:

- Three logical components: web server, processor & notifier
- Each component is a separate package
- Package main takes care of flag parsing and booting the chosen component
- Single binary with subcommands for choosing component to boot

```
● downloader@notifier.service - Downloader Server (notifier)
   Loaded: loaded (/etc/systemd/system/downloader@.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2018-07-03 12:59:23 EEST; 7 months 24 days ago
     Docs: https://github.skroutz.gr/skroutz/downloader
 Main PID: 24457 (downloader)
   CGroup: /system.slice/system-downloader.slice/downloader@notifier.service
           └─24457 /usr/local/lib/downloader/bin/downloader notifier -c /etc/downloader/config.json

Feb 25 16:18:14 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:14 Performing callback request for Job{ID:UEjwx4Irgt
Feb 25 16:18:14 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:14 Performing callback request for Job{ID:hRvuIRld4i
Feb 25 16:18:53 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:53 Performing callback request for Job{ID:769sNJmqRS
Feb 25 16:18:53 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:53 Performing callback request for Job{ID:7K3eHXXS_d
Feb 25 16:18:53 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:53 Performing callback request for Job{ID:BbhOmahdvx
Feb 25 16:18:53 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:53 Performing callback request for Job{ID:IW1ff6nuVy
Feb 25 16:18:53 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:53 Performing callback request for Job{ID:Jp1RpeB2w8
Feb 25 16:18:53 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:53 Performing callback request for Job{ID:QCOSpqQz5p
Feb 25 16:18:53 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:53 Performing callback request for Job{ID:ZMmd5fQmha
Feb 25 16:18:53 dl1 downloader[24457]: [notifier] 2019/02/25 16:18:53 Performing callback request for Job{ID:c1ruOetFfp

● downloader-purge-assets.service - Purge fetched downloader assets
   Loaded: loaded (/etc/systemd/system/downloader-purge-assets.service; enabled; vendor preset: enabled)
   Active: active (running) since Fri 2018-05-04 16:11:56 EEST; 9 months 23 days ago
 Main PID: 548 (sh)
    Tasks: 4 (limit: 4915)
   Memory: 15.0M
   CGroup: /system.slice/downloader-purge-assets.service
           ├─548 /bin/sh -c /usr/bin/tail --follow=name $FILE | /usr/local/lib/downloader/bin/purge-assets --purge
           ├─556 /usr/bin/tail --follow=name /var/log/nginx/downloader.access.log
           └─557 ruby /usr/local/lib/downloader/bin/purge-assets --purge

Feb 25 06:25:06 dl1 sh[548]: /usr/bin/tail: /var/log/nginx/downloader.access.log: No such file or directory
```

downloader

Lessons learned:

- Break logical entities into separate packages
- net/http defaults will likely not be a good fit for production
  - TLS options (misconfigured upstreams)
  - Timeouts (see "The Complete Guide to net/http Timeouts")

Available at https://github.com/skroutz/downloader/

mistry

Project #4

mistry

Background:

- We test ~50 times and deploy ~40 times per day
- Increasingly *slow test-deploy cycles*
- Good candidate problem
  - Well-defined
  - Purely technical, no business decisions

mistry

At a glance:

- A generic build server
- Speeds up builds
  - Caches and re-uses results
  - Incremental building
  - ...and others

mistry

Why Go:

1. Concurrency
    a. Multiple builds in parallel
    b. Serve concurrent clients
2. Interface with Docker

mistry

Benefits:

1. Build pipelines sped up significantly
    a. Deploy times went from 16 mins. down to 3 mins.
    b. ~3 minutes shaved off our test suite
2. People may download artifacts for diagnostic reasons
3. Out of the box statistics for every pipeline

mistry

Lessons learned:

- Go makes end-to-end testing easy
  - `go test` compiles an actual binary you can use right in your tests
  - Utility: https://github.com/agis/spawn/

Available at https://github.com/skroutz/mistry

Idioms, Tools and Practices

Idioms

- Avoid naked returns
- Use Context *only* for cancellation in goroutine *hierarchies*
- package types
- Table-driven tests

Idioms

- Ship things as Debian packages
- Expose a metrics endpoint
- Log to stdout and stderr
- Wishlist: Functional options


RUNTIME
Goroutines
20
15
10
5
0
18:00
18:10
18:20
18:30
18:40
18:50
Heap size
95 MiB
76 MiB
57 MiB
38 MiB
19 MiB
0 B
18:00
18:10
18:20
18:30
18:40
18:50
min
max
avg
HeapAlloc 40.29 MiB 75.07 MiB 56.79 MiB
HeapInuse 45.47 MiB 76.57 MiB 59.83 MiB
Heap objects
1.0 Mil
33 K
1 K
32
1
0
18:00
18:10
18:20
18:30
18:40
18:50
min
max
avg
HeapObjects 19 K 394 K 197 K
Heap objects freed
1.5 Mil
1.0 Mil
500 K
0
18:00
18:10
18:20
18:30
18:40
18:50
avg
Frees 1.122 Mil
Mallocs 1.123 Mil
GC pauses
4.0 ms
3.0 ms
2.0 ms
1.0 ms
0 ns
18:00
18:10
18:20
18:30
18:40
18:50
max
avg
total
scratch1.PauseTotalNs 3.1 ms 461.0 µs 27.2 ms
GC cycles
4.0
3.0
2.0
1.0
0
18:00
18:10
18:20
18:30
18:40
18:50
avg
NumGC 3
Stack size
854 KiB
830 KiB
806 KiB
781 KiB
757 KiB
732 KiB
18:00
18:10
18:20
18:30
18:40
18:50
min
max
avg
current
StackInuse 736.00 KiB 832.00 KiB 786.13 KiB 832.00 KiB
StackSys 736.00 KiB 832.00 KiB 786.13 KiB 832.00 KiB

Our tools of choice

- Mirror for internal packages
  - `go get go.skroutz.gr/scratchd`
  - How to: https://golang.org/cmd/go/#hdr-Remote_import_paths
- errcheck
- Building: Makefiles (example)
- Dependency management: dep (not for long though)
- Logging: package log

Our tools of choice

- Configuration: package encoding/json
- Metrics: package expvar
- Performance analysis: `go tool pprof`
- Runtime visibility: `go tool trace`

Best Practices

- Always check for errors
- Avoid package level variables
- Run load tests with the race detector enabled!
- Write end-to-end tests for critical systems
- Monitor number of goroutines and heap used (runtime.MemStats)
- Integrate `gofmt -s` and `go vet` into your CI
- Follow https://golang.org/doc/effective_go.html

Takeaways

Takeaways

When considering to introduce a new technology in your org:

- Pick the right problem
- Prototype first
- Keep it small and simple

Takeaways

If you are considering or just picked Go:

- Goroutines & channels doesn’t mean you’ll avoid all the races
  - Don’t assume goroutine-safety
  - The race detector is your friend
- Think of when your goroutines terminate and how you’ll notice
- The standard library has most of the things you’ll need

Takeaways

If you are considering Go (cont.):

- Invest time in learning the tools
- CGo is a different beast
  - More at "CGo is not Go"
- golang.org/doc/ is your friend

Questions?

Thank you!

agis.io @agisanast github.com/agis