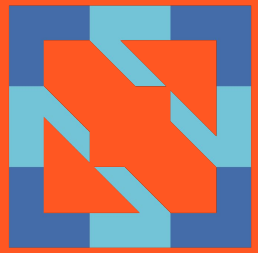
Rust Flavored Cloud-Native

Sabree Blackmon
@HeavyPackets
Senior Security Engineer, Docker

Thanks to the CNCF for sponsoring



CLOUD NATIVE
COMPUTING FOUNDATION

Thanks to 1904Labs for the space

What is Cloud-Native?

What is Cloud-Native?

CNCF Cloud Native Definition v1.0

“Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds.” - CNCF

What is Cloud-Native?

CNCF Cloud Native Definition v1.0

Techniques that enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow teams to make high-impact changes frequently and predictably without pain - CNCF

In practice, this means...

- Containers
- Service Meshes
- Microservices
- Immutable Infrastructure
- Declarative APIs
- Distributed Observability
  - Logging
  - Metrics
  - Tracing

What is 12-Factor?

The Twelve-Factor App

- 12-factor is a *guidance system* for devs and ops for deploying apps that are portable, well-behaved, predictable and scalable
- As such, it's a **great but imperfect** starting point for building cloud-native applications

12-Factor & Rust

I. Codebase

One codebase tracked in revision control, many deploys

- Use a DVCS like Git
- All relevant Rust project config is stored in Cargo.toml

II. Dependencies

Explicitly declare and isolate dependencies

- Check in your Cargo.lock to ensure version stability across platforms
- Use a tool like cargo-edit to manage your crate dependencies
- Use a Cargo build-script to encapsulate pre-build dependencies

III. Config

Store config in the environment

- Use a crate like envconfig for type-safe environmental variable parsing
- Use lazy_static to safely make app config *everywhere* available without cloning

IV. Backing services

Treat backing services as attached resources

- Load connection strings/URIs as runtime configurations
- Write proper try & failure behavior around all external app connectivity

V. Build, release, run

Strictly separate build and run stages

- Avoid use of conditional compilation based on runtime targets
- If possible, allow app to be configured at runtime for different platforms

VI. Processes

Execute the app as one or more stateless processes

- Eagerly share state between app instances via KV-store or in-memory database
- Use serde to enable typed serialization to JSON which many KV-stores can use

VIII. Concurrency

Scale out via the process model

- Highly **concurrent** code is great use of CPU time
  - Tokio for async IO
- Highly **parallelizable** work may be better scaled out across processes
  - Rayon allows for dead simple parallelized iterator operations
  - Work can be stolen across threads but is constricted by per process cpu/mem limits

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

- Encapsulate startup tasks into type compatible futures so that they can be executed concurrently
- Join futures & map errors after all tasks have run
- Use scopeguard to allow panic safe RIAA

X. Dev/prod parity

Keep development, staging, and production as similar as possible

- Build your app inside of containers to prevent environmental taint
- Keep amount of runtime configuration required low to prevent drift
  - Allow for sane defaults

XI. Logs

Treat logs as event streams

- Use log crate for stdout/stderr logging
- Implement events in types for structured logging or consistent printing (as text)

XI. Logs

Treat logs as event streams

- Use fruently to for distributed logging of events via fluentd & others
- Ensure logging functionality does not block
  - Use threads and unbounded channels for greater efficiency

XII. Admin processes

Run admin/management tasks as one-off processes

- Create API services/endpoints for administrative tasks
  - Protect these endpoints with mTLS/authZ
- Use an ORM, like Diesel, for versioned DB migrations

XII. Admin processes

Run admin/management tasks as one-off processes

- Use subprocess crate for multidirectional system exec for node level interaction
- Wrap system level interaction with functions and types to increase safety and portability
  - Write integration tests assertions to protect against regression if command interface changes
  - Wrap native libraries if possible

Where to go from here?

What about?

- Communication protocols in Rust
  - HTTP
  - GRPC + Protobuf
- Security
  - HTTPS & mutual TLS
  - authZ
- Metrics
- Storage

Questions?

@HeavyPackets

That's all folks!