The Rusting Cloud

Krustlet, Kubernetes, and Building Apps for the Cloud

Taylor Thomas

Who am I?

e.g. Why my opinion might matter

- Krustlet and Helm Core Maintainer
- Doing Kubernetes things since 1.2 and Docker since 0.7
- Rustacean by way of Go
- Social media handles
  - Twitter: @_oftaylor
  - GitHub: @thomastaylor312
  - Kubernetes Slack: @oftaylor

Agenda

- A brief intro to some projects (and Wasm)
- But why?
- Rust in Cloud Development
  - The Good
  - The Bad
  - The Ugly
- Bonus cage fight: Rust vs. Go

Project Intro

WebAssembly

Better known as Wasm

- Wasm is a compiled binary that can be run in a browser through Javascript
- WASI: WebAssembly System Interface
  - A standard for interacting with a host system, no matter the OS
  - It is relatively new and not fully fleshed out yet, but is on its way there

The Projects

- Krustlet: https://github.com/deislabs/krustlet
- Bindle: https://github.com/deislabs/bindle
- WAGI: https://github.com/deislabs/wagi
- WASI HTTP support: https://github.com/deislabs/wasi-experimental-http

Projects

Krustlet: https://github.com/deislabs/krustlet

- Kubernetes RUST kubeLET
- Its primary purpose is to run Wasm modules within Kubernetes
- Multiple Providers
- 3 Rust Crates
  - Kubelet
  - Krator: https://deislabs.io/posts/introducing-krator/
  - OCI-Distribution

Projects

Bindle: https://github.com/deislabs/bindle

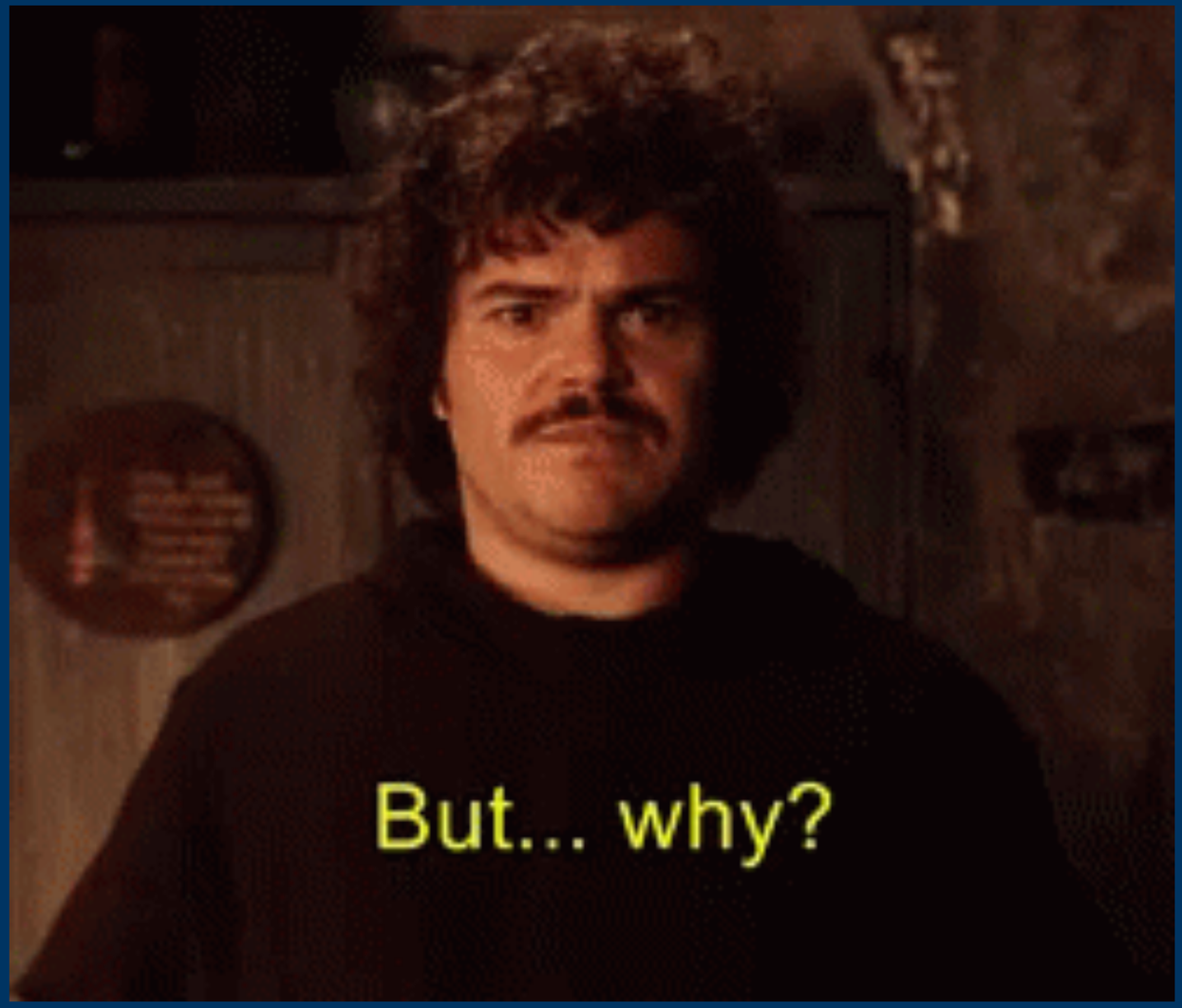
- Aggregate Object Storage
- The first use case is for enabling our WASM/WASI projects
- Turns out it could be really useful for securing software supply chains

Projects

WAGI: https://github.com/deislabs/wagi

- WebAssembly Gateway Interface
- A (partial) CGI implementation that allows WASI modules to act as HTTP handlers (similar to FaaS)
- Another way to enable HTTP for WASI

Why?



Wasm and WASI

But I already have containers!

- Security
- Density
- Actually “run everywhere”
- Smaller footprint for embedded devices

Rust and Wasm are friends

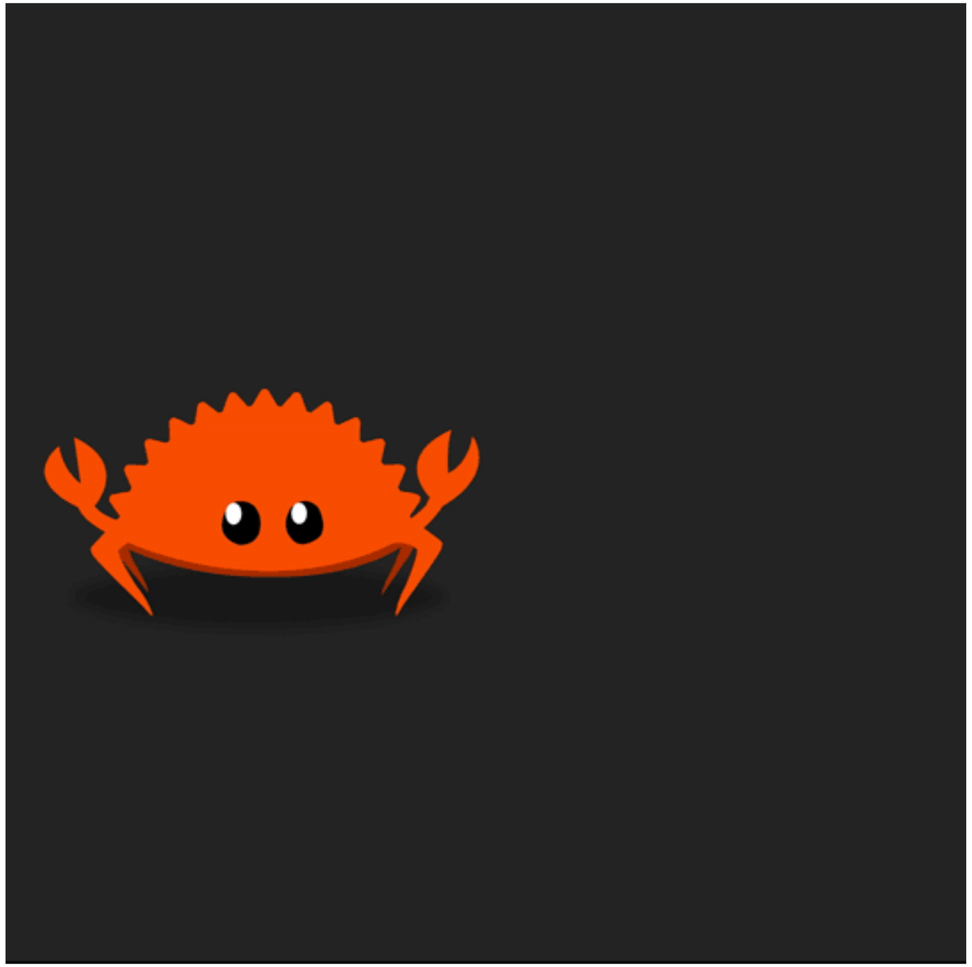
WEBASSEMBLY

Kubernetes and the Cloud

- For many people (especially late adopters), Kubernetes == Cloud
- Krustlet was created to allow people to try Wasm in a familiar environment
- We discovered that Rust was actually a *great* fit for working with Kubernetes and other cloud applications

Rust

- Safety
- Developer Experience
- Wasm Support
- Extensibility



@_oftaylor

Rust In Cloud Development

https://deislabs.io/posts/still-rusting-one-year-later/

Rust In Cloud Development

The Good

The Good

Traits

```
async fn get_yanked_invoice<I>(&self, id: I) -> Result<super::Invoice>
    where
        I: TryInto<Id> + Send,
        I::Error: Into<ProviderError>;
```

The Good

Enums

```rust
pub enum ClientError {
    /// The item already exists
    AlreadyExists,
    /// The error returned when the request is invalid. Contains the underlying HTTP status code and
    /// any message returned from the API
    InvalidRequest {
        status_code: reqwest::StatusCode,
        message: Option<String>,
    },
    /// A server error was encountered. Contains an optional message from the server
    ServerError(Option<String>),
}

pub fn handle_error(e: ClientError) {
    match e {
        ClientError::AlreadyExists => {
            println!("Item already exists")
        }
        ClientError::InvalidRequest { status_code, message } => {
            println!("Invalid request. HTTP code: {}, message: {}", status_code, message.unwrap_or_default())
        }
        ClientError::ServerError(Some(message)) => {
            println!("Server error: {}", message)
        },
        ClientError::ServerError(None) => {
            println!("Server error")
        },

    }
}
```

@_oftaylor

The Good

Enums

```
pub enum Transition<S: ResourceState> {
    /// Transition to new state.
    Next(StateHolder<S>),
    /// Stop executing the state machine and report the result of the execution.
    Complete(anyhow::Result<()>),
}
```

@_oftaylor

The Good

Macros

```
#[derive(CustomResource, Debug, Serialize, Deserialize, Clone, Default, JsonSchema)]
#[kube(
    group = "animals.com",
    version = "v1",
    kind = "Moose",
    derive = "Default",
    status = "MooseStatus",
    namespaced
)]
struct MooseSpec {
    height: f64,
    weight: f64,
    antlers: bool,
}
```

@_oftaylor

The Good

Error Handling and Iterators

```
let missing = inv
    .parcel
    .as_ref()
    .unwrap_or(&zero_vec)
    .iter()
    .map(|k| async move {
        let parcel_path = self.parcel_path(k.label.sha256.as_str());
        // Stat k to see if it exists. If it does not exist or is not a directory, add it.
        let res = tokio::fs::metadata(parcel_path).await;
        match res {
            Ok(stat) if !stat.is_dir() => Some(k.label.clone()),
            Err(_e) => Some(k.label.clone()),
            _ => None,
        }
    });

Ok(futures::future::join_all(missing)
    .await
    .into_iter()
    .filter_map(|x| x)
    .collect())
```

The Good

Dependency Management

```
k8s-openapi = { version = "0.11", default-features = false, features = ["v1_18"] }
structopt = { version = "0.3", features = ["wrap_help"], optional = true }
lazy_static = "1.4"
oci-distribution = { path = "../oci-distribution", version = "0.5", default-features = false }
```

```
#[cfg(any(feature = "cli", feature = "docs"))]
#[cfg_attr(feature = "docs", doc(cfg(feature = "cli")))]
pub struct Opts
```

@_oftaylor

The Good

Community

@_oftaylor

Rust In Cloud Development

The Bad

The Bad

Docs and Clarity


```rust
Endpoint::from_static("http://[::]:50051")
    .connect_with_connector(service_fn(move |_: Uri| {
        // Connect to a Uds socket
        UnixStream::connect(p.clone())
    }))
    .await;
```

@_oftaylor

The Bad

Missing Crate Functionality

```
impl Stream for Socket {
    type Item = Result<UnixStream, std::io::Error>;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>> {
        futures::ready!(self.listener.poll_read_ready(cx, Ready::readable()))?;

        let stream = match self.listener.get_ref().accept() {
            Ok(None) => {
                self.listener.clear_read_ready(cx, Ready::readable())?;
                return Poll::Pending;
            }
            Ok(Some((stream, _))) => stream,
            Err(_) => {
                self.listener.clear_read_ready(cx, Ready::readable())?;
                return Poll::Pending;
            }
        };
        let mut fut = UnixStream::new(stream).boxed();
        let stream = loop {
            match fut.poll_unpin(cx) {
                Poll::Ready(res) => break res,
                Poll::Pending => continue,
            }
        };
        Poll::Ready(Some(stream))
    }
}
```

```
pub mod skinny {
    use mio;
    use std::sync::{atomic::AtomicUsize, Arc, Condvar, Mutex};

    fn reinterpret_cast<T, U>(obj: &T) -> &U {
        unsafe { &*(obj as *const T as *const U) }
    }

    pub mod sys {
        use super::reinterpret_cast;
        use lazycell::AtomicLazyCell;
        use mio::windows::Binding as MiowBinding;
        use miow::iocp::CompletionPort;
        use std::sync::{Arc, Mutex};

        #[derive(Debug)]
        struct Binding {
            selector: AtomicLazyCell<Arc<SelectorInner>>,
        }
```

@_oftaylor

The Bad

The Learning Curve

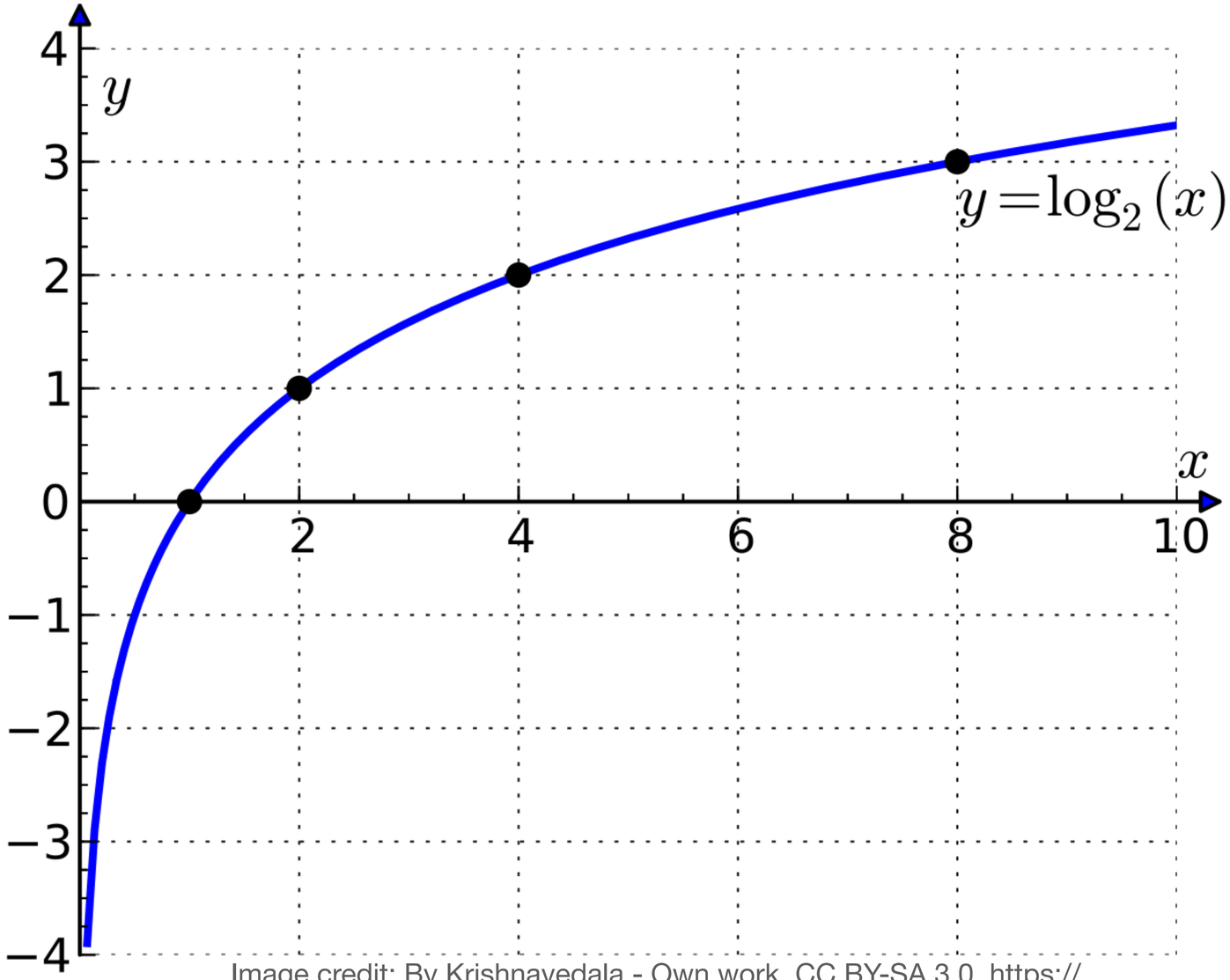


Image credit: By Krishnavedala - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15408195

@_oftaylor

Rust In Cloud Development

The Ugly

The Ugly

Competing and Incompatible Runtimes

- Multiple runtimes (tokio, async-std, smol, etc)
- Lock-in once you choose a runtime

The Ugly

Ugly Code and Nested Futures

```
let mk_svc = make_service_fn(move |conn: &AddrStream| {
    let addr = conn.remote_addr();
    let r = router.clone();
    async move {
        Ok::<_, std::convert::Infallible>(service_fn(move |req| {
            let r2 = r.clone();
            async move { r2.route(req, addr).await }
        }))
    }
});
```

```
let signal_task = start_signal_task(Arc::clone(&signal)).fuse().boxed();

let plugin_registrar = start_plugin_registry(self.provider.plugin_registry())
    .fuse()
    .boxed();

let webserver = start_webserver(self.provider.clone(), &self.config.server_config)
    .fuse()
    .boxed();

let node_updater = start_node_updater(client.clone(), self.config.node_name.clone())
    .fuse()
    .boxed();

let services = Box::pin(async {
    tokio::select! {
        res = signal_task => if let Err(e) = res {
            error!("Signal task completed with error {:?}", &e);
        },
        res = webserver => error!("Webserver task completed with result {:?}", &res),
        res = node_updater => if let Err(e) = res {
            error!("Node updater task completed with error {:?}", &e);
        },
        res = plugin_registrar => if let Err(e) = res {
            error!("Plugin registrar task completed with error {:?}", &e);
        }
    };
    signal.store(true, Ordering::Relaxed);
    Ok::<(), anyhow::Error>(())
});
```

@_oftaylor

The Ugly

Cruft and Bloat

```rust
impl Connected for UnixStream {}

impl AsyncRead for UnixStream {
    fn poll_read(
        mut self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        buf: &mut ReadBuf<'_>,
    ) -> Poll<std::io::Result<()>> {
        Pin::new(&mut self.0).poll_read(cx, buf)
    }
}

impl AsyncWrite for UnixStream {
    fn poll_write(
        mut self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        buf: &[u8],
    ) -> Poll<std::io::Result<usize>> {
        Pin::new(&mut self.0).poll_write(cx, buf)
    }

    fn poll_flush(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<std::io::Result<()>> {
        Pin::new(&mut self.0).poll_flush(cx)
    }

    fn poll_shutdown(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<std::io::Result<()>> {
        Pin::new(&mut self.0).poll_shutdown(cx)
    }
}
```

```rust
pub async fn raw<P: AsRef<Path>>(
    file_path: P,
) -> Result<impl Stream<Item = std::result::Result<bytes::BytesMut, Error>>> {
    let file = File::open(file_path).await?;
    Ok(FramedRead::new(file, BytesCodec::new()))
}
```

@_oftaylor

Rust vs Go

Rust vs Go

What do we miss from Go?

- Ease of concurrent task scheduling
- A "batteries included" standard library, including HTTP support
- Easier to onboard new developers

Rust vs Go

What do we NOT miss about Go?

- Dependency management
- if err != nil
- Lack of generics (even with the newly added ones)
- Governance model
- Lack of collections and iterator functions
- Lack of safety

Rust vs Go

When to use each one

Go

- Small/quick projects
- True microservices
- Super deep integration with specific cloud tools (e.g. Kubernetes, Docker)

Rust

- Larger projects
- Flexible APIs
- Safety or Security Requirements

Questions?

Thank You!