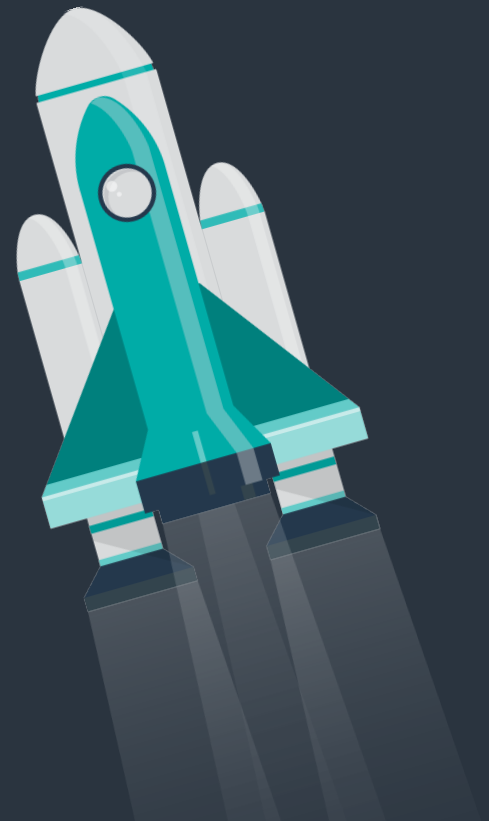
Runtime Scripting for Rust Applications

Niklas Korz

About Me

- Co-founder and tech lead at alugha
- MSc Data and Computer Science graduate
- Website: https://korz.dev
- Mastodon: @niklaskorz@rheinneckar.social
- Interests: Rust, Nix, Distributed Systems, Natural Languages, ...

1. Motivation
2. Languages and Implementations
3. Embedding Deno
4. Further Reading



Rust is...

- statically typed
- compiled ahead of time
- borrow checked

When to Script

- Fast prototyping
- Allow end-users to change runtime behaviour
  - C-ABI? Stable Rust ABI?
  - Sandboxed execution
- Hot reloading
- Collaborate with people who find Rust too complicated

Rust is...

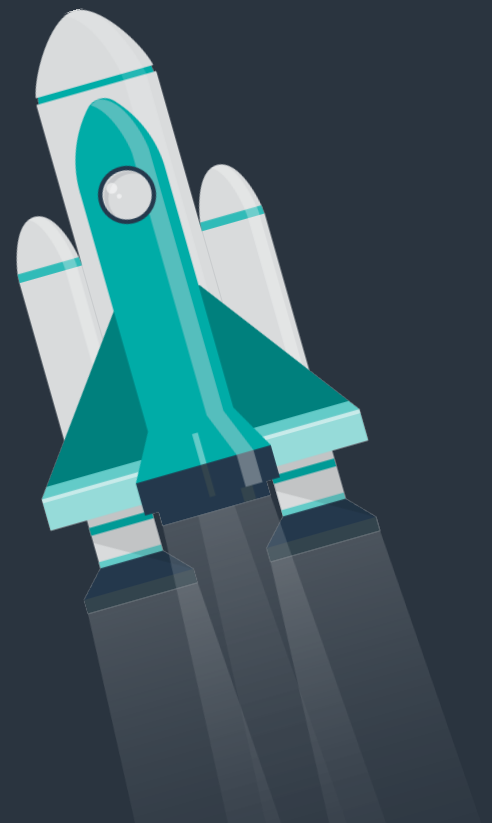
- statically typed
- compiled ahead of time
- borrow checked

Python* is...

- dynamically typed 🦆
- interpreted
- garbage collected

* and JavaScript, Lua, Ruby, ...

1. Motivation
2. **Languages and Implementations**
3. Embedding Deno
4. Further Reading



WebAssembly

- Portable binary code, usually JIT-compiled
- Stable ABI
- Sandboxed execution
- Cross-language
  - but you have to bring your own allocator
  - except if you're using garbage collection

WebAssembly

- Supports SIMD128 vectorisation
- WebAssembly System Interface
  - Filesystem access, network sockets, ...
- WebAssembly + Rust = 💕
  - Wasmtime by Bytecode Alliance
  - Wasmer

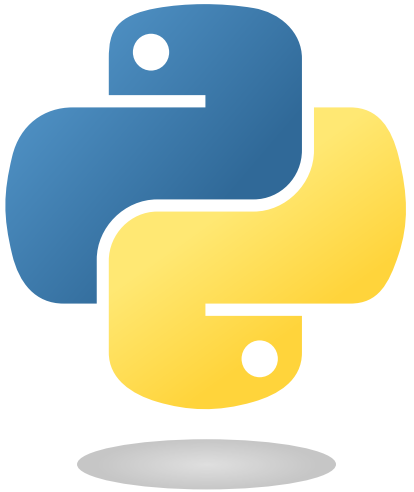
Built for Rust

- Mun:
  - AOT compiled, statically typed
  - Designed for hot reloading
  - LLVM for compilation and optimisation
- Rhai:
  - AST-interpreter (relatively slow)
  - Dynamically typed
  - Custom operators

Lua

- Fast: stack-based interpreter or JIT compiled
- Lightweight (code: 355kb Lua vs 25mb CPython vs 37mb v8)
- mlua for Rust:
  - wraps C Lua implementations (Lua/LuaJIT/Luau)
  - async/await support
  - sandboxing (with Luau)

Python



- Bytecode interpreter (at least in CPython)
- optional static typing
- PyO3: wraps CPython, often used for libraries
- RustPython: pure Rust implementation, (yet) incomplete
- Both support async/await

ECMAScript*

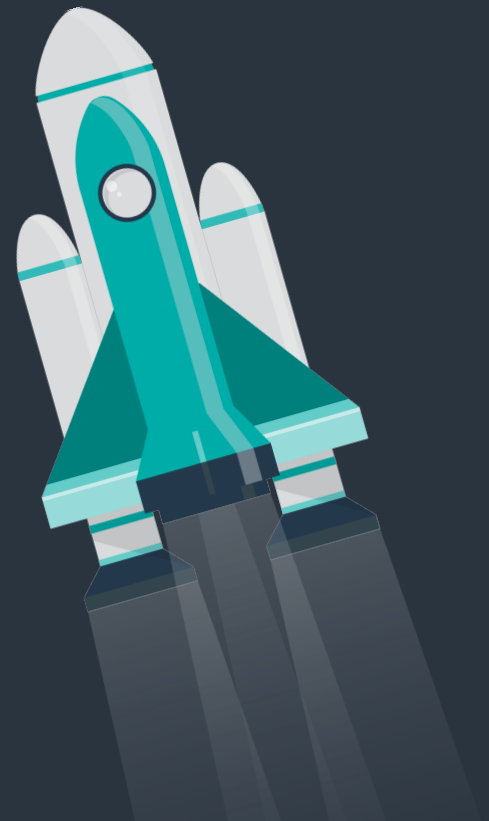
* aka JavaScript or TypeScript

- Standardised (ECMA-262)
- Optional static typing (almost standard)
- Fast: mix of interpreter and JIT
- Multiple major implementations:
  - Gecko (Firefox): SpiderMonkey
  - WebKit (Safari): JavaScriptCore
  - Blink (Chrome): v8

ECMAScript in Rust

- Deno: v8-based Node.js competitor
  - mature and stable
  - implements many Web APIs like fetch, localStorage
- rusty_jsc: based on JavaScriptCore
  - relatively young, easy to use
- mozjs: based on SpiderMonkey, used in Servo
  - complicated API, requires use of `unsafe` C-APIs

1. Motivation
2. Languages and Implementations
3. **Embedding Deno**
4. Further Reading



Embedding Deno in Rust

- deno_core:
  - wraps v8 in a safe Rust API (rusty_v8)
  - provides extension mechanism "ops"
  - implements serde for v8 objects (beware of UTF-16)
  - import hooks (`ModuleLoader`)

Embedding Deno in Rust

- deno_ast:
  - transpiler for TypeScript, JSX and ES proposals
  - based on SWC (Rust alternative to Babel)
  - extendible: SWC plugins in WebAssembly or ES

Embedding Deno in Rust

- deno_runtime:
  - batteries included
  - implements Web APIs such as:
    - fetch, localStorage, WebGPU, Web Workers, Web Sockets
  - additional APIs:
    - filesystem access, HTTP server, TCP/UDP sockets

Exposing APIs from Rust to Deno

- Deno Extensions: provide native "ops" and some JS glue
  - ops only accessible inside the JS glue code (`Deno.core`)
  - sync ops: `Deno.core.ops.opName(...args)`
  - async: `Deno.core.opAsync("opName", ...args)`
- serde_v8: converts between JS types and Rust types

```rust
pub struct HostState {
    pub n: i32,
}
let host_state = Arc::new(RwLock::new(
  HostState { n: 42 },
));
{
    let state = worker.js_runtime.op_state();
    let mut state = state.borrow_mut();
    state.put(host_state);
}
```

```rust
#[op2(async)]
pub async fn op_scripting_test(
  state: Rc<RefCell<OpState>>, n: i32
) -> i32 {
    let lock = state.borrow()
        .borrow::<Arc<RwLock<HostState>>>().clone();
    let mut host_state = lock.write().await;
    host_state.n += n;
    host_state.n
}
```

```
const { core } = Deno;

globalThis.opScriptingTest = (...args) =>
  core.opAsync("op_scripting_test", ...args);
```

```
extension!(
    my_extension,
    ops = [op_scripting_test],
    esm_entry_point = "ext:my_extension/my_extension.js",
    esm = ["my_extension.js"],
    docs = "A small sample extension"
);
```

```ts
export function addToHostState(
  n: number
): Promise<number> {
  return globalThis.opScriptingTest(n);
}
```

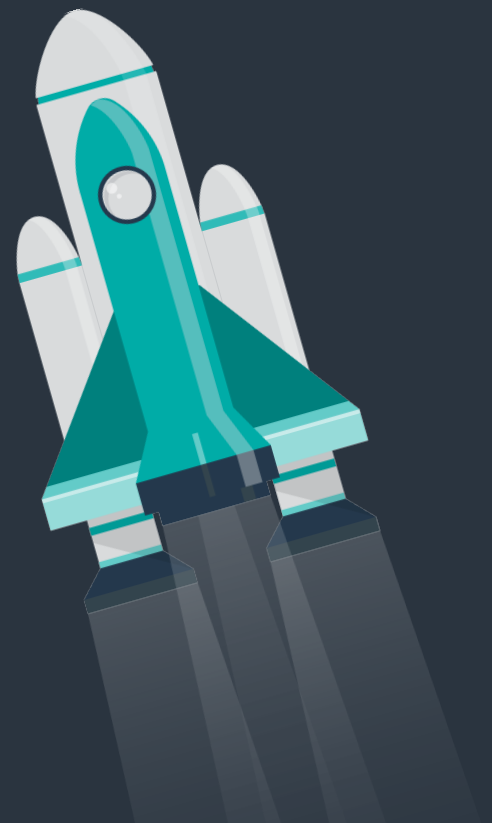
```ts
import { sum, addToHostState } from "builtin:math";

export async function test() {
  console.log("hello world", sum(1, 2));

  const response = await fetch("https://api.ipify.org/");
  const text = await response.text();
  console.log("IP:", text);

  console.log("New host state:", await addToHostState(2));
  console.log("New host state:", await addToHostState(3));
}
```

1. Motivation
2. Languages and Implementations
3. Embedding Deno
4. **Further Reading**



Further Reading

- Example: https://codeberg.org/niklaskorz/rust-scripting-example
- Deno: Roll your own JavaScript runtime (three parts)
- Deno: Internals of Deno
- mlua: Bevy Scripting with Lua
- mlua: Lua in the Browser, with Rust and WebAssembly
- PyO3: Calling Python in Rust Code

Questions?

- Download: https://dl.korz.dev/meetup-rust-scripting.pdf (.dev, not .de!)
- Contact me on…
  - Matrix: @niklaskorz:matrix.org
  - Mastodon: @niklaskorz@rheinneckar.social
  - Email: contact@korz.dev