methodpark
by UL

# Share Business Logic in Rust across Languages and Platforms

Rust Franken Meetup #3

2021-11-18

Bernd Kaiser

https://github.com/meldron/shared_rust

# About me

Bernd Kaiser

Senior Software Engineer

MobileApps & WebApps

Method Park by UL

# Agenda

1. Motivation
2. C FFI
3. wasm-pack
4. PyO3
5. uniffi-rs
6. flutter_rust_bridge

# Motivation

Typical software stack consists of many programming languages which target many systems/OSs.

➜ Same functionality implemented many times.

# Rust to the rescue 

Write/test business logic once, call from everywhere™.

# Why Rust?

- Rust compiles to many targets
- Safe & modern language
- Great crates which simplify FFI setups

# Our Business Logic

```rust
use unicode_normalization::UnicodeNormalization;
use unicode_security::confusable_detection::skeleton;

pub fn normalize(s: &str) -> String {
    let unconfused: String = skeleton(&s).collect();
    let upper = unconfused.to_uppercase();
    let normalized: String = upper.nfkd().collect();

    normalized
}
```

Example:

```rust
normalize_username("𝐵𝑒𝕣𝕟𝕕") == "BERND"
```

Unicode Normalization Forms - Unicode Confusable Detection

# C FFI

The great connector

# C FFI

Almost all programming languages can be extended via the C FFI.

Rust is no exception.

```rust
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(l: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length: {}", x);
}
```

# Rust from C

We need a header file and a shared library or static library to link against.

➜ Create a Wrapper Crate that exports the needed functions so that the C compiler/linker can call them.

# Business Logic Wrapper Crate - `Cargo.toml`

```toml
[package]
edition = "2021"
name = "norm"
version = "0.1.0"

[lib]
# cdylib ➜ dynamic system library (.so|.dll|.dylib)
# staticlib ➜ static system library  (.a|.lib)
crate-type = ["cdylib", "staticlib"]
name = "norm"

[dependencies]
shared = {path = "../shared"}
[build-dependencies]
cbindgen = "0.20.0"
```

# Business Logic Wrapper Crate - `lib.rs`

```rust
use shared::normalize_username as normalize;

use std::{
    ffi::{CStr, CString},
    os::raw::c_char,
};

#[no_mangle]
pub extern "C" fn normalize_username(p: *const c_char) -> *const c_char {
    let raw = unsafe { CStr::from_ptr(p) };
    let s = raw.to_str().expect("invalid utf-8");

    let normalized = normalize(s);

    let c_string = CString::new(normalized).expect("could not build c string");

    let ptr = c_string.as_ptr();
    std::mem::forget(c_string);

    ptr
}
```

Foreign calling conventions

# Create c header files

`cbindgen` creates C/C++11 headers for Rust libraries which expose a public C API

```
cbindgen --lang c . -o 'example/normalizer.h'
```

Creates:

```
1 #include <stdarg.h>
2 #include <stdbool.h>
3 #include <stdint.h>
4 #include <stdlib.h>
5
6 const char *normalize_username(const char *p);
```

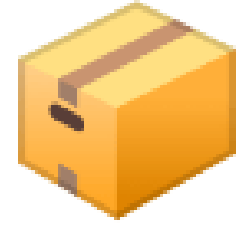
# C BUILD FII DEMO

```
# build dynamically linked executable
cargo build --release
cp target/release/libnorm.so example
gcc main.c -o normalize -I. -L. -l:libnorm.so

# build statically linked executable
cargo build --release --target x86_64-unknown-linux-musl
cp target/x86_64-unknown-linux-musl/release/libnorm.a \
    example/libnorm.a
musl-gcc main.c -o snormalize -static -I. -L. -l:libnorm.a
```

# C FFI Links

- https://github.com/eqrion/cbindgen
- https://michael-f-bryan.github.io/rust-ffi-guide/overview.html - FFI Overview
- https://github.com/getditto/safer_ffi - Framework that helps you write FFI
- https://doc.rust-lang.org/reference/linkage.html - `crate-types` explained

# wasm-pack 

one-stop shop for building and working with Rust generated WebAssembly

# Why do you should use wasm-pack

- Rust supports target `wasm32-unknown-unknown` out of the box
- But it will only create a single `.wasm` library without any JavaScript glue code
- Creating this glue code is not trivial and also `.d.ts` files would have to be created

# wasm-pack to the rescue

## Simple installation

```
cargo install -f wasm-pack
wasm-pack new wasm
```

## Initializes a wasm project from a template:

```
./wasm:              ./wasm/src:       ./wasm/tests:
Cargo.toml           lib.rs            web.rs
LICENSE_APACHE       utils.rs
LICENSE_MIT
README.md
src
tests
```

# wasm-pack - Cargo.toml

```toml
[lib]
crate-type = ["cdylib", "rlib"]

[features]
default = ["console_error_panic_hook", "wee_alloc"]

[dependencies]
wasm-bindgen = "0.2.63"
# The `console_error_panic_hook` crate provides better debugging of panics by
# logging them with `console.error`. This is great for development, but requires
# all the `std::fmt` and `std::panicking` infrastructure, so isn't great for
# code size when deploying.
console_error_panic_hook = { version = "0.1.6", optional = true }
# `wee_alloc` is a tiny allocator for wasm that is only ~1K in code size
# compared to the default allocator's ~10K. It is slower than the default
# allocator, however.
wee_alloc = { version = "0.4.5", optional = true }

[profile.release]
# Tell `rustc` to optimize for small code size.
opt-level = "s"
```

# wasm-pack - lib.rs

```rust
use shared::normalize_username as normalize;
use wasm_bindgen::prelude::*;

#[cfg(feature = "wee_alloc")]
#[global_allocator]
static ALLOC: wee_alloc::WeeAlloc =
    ee_alloc::WeeAlloc::INIT;

#[wasm_bindgen]
pub fn normalize_username(s: &str) -> String {
    utils::set_panic_hook();
    normalize(s)
}
```

# wasm-pack build

```
wasm-pack build
# wasm-pack build --target nodejs -d node
```

creates ./pkg directory with:

```
package.json
README.md
wasm_bg.js
wasm_bg.wasm
wasm_bg.wasm.d.ts
wasm.d.ts
wasm.js
```

which can be published to npm.

# How to use wasm package from the browser

- `wasm-pack` provides a npm template with a webpack template
- `npm init wasm-app example` creates a new directory with a webpack setup to bootstrap the wasm library

# How to use wasm package from node

```
> const normalizer = require('./wasm');
> w.normalize_username("Rⓤ(S)ⓣ🦀")
'RU(S)T🦀'
```

# wasm demo

# Rust `Strings` vs `JsStrings`

- Rust `Strings` are `UTF-8`
- `JsStrings` are `UTF-16` where unpaired surrogates are allowed

# Use `JsString`s in your wasm

```rust
use js_sys::JsString; // needs js-sys crate

#[wasm_bindgen]
pub fn normalize_username_js_string(s: JsString)
-> Result<JsString, JsValue> {
    utils::set_panic_hook();
    if !s.is_valid_utf16() {
        return Err(JsValue::from_str("Invalid utf-16"));
    }

    let normalized = normalize(String::from(s).as_str());

    Ok(JsString::from(normalized))
}
```

# Useful Links

- https://github.com/rustwasm/wasm-pack
- https://github.com/rust-lang/rust-bindgen
- https://docs.rs/js-sys
- https://github.com/neon-bindings/neon (NodeJS)

# PyO3

- Rust bindings for Python
- Tools for creating native Python extension modules
- Tools for distributing Python wheels

# Python Native Extensions

```c
#include <Python.h>

static PyObject * normalize_username(PyObject *self, PyObject *args) {
    // ...
}

static PyMethodDef methods[] = {
    {
        "normalize_username", normalize_username, METH_VARARGS, "Normalizes a username."
    },
};

static struct PyModuleDef normalizer_definition = {
    PyModuleDef_HEAD_INIT, "normalizer",
    "normalize your usernames", -1, methods
};

PyMODINIT_FUNC PyInit_normalizer(void)
{
    Py_Initialize();
    return PyModule_Create(&normalizer_definition);
}
```

Extending Python with C or C++

# PyO3 to the rescue

```rust
use pyo3::prelude::*;
use shared::normalize_username as normalize;

#[pyfunction]
fn normalize_username(s: &str) -> String {
    normalize(s)
}

#[pymodule]
fn normalizer(py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(normalize_username, m)?)?;
    Ok(())
}
```

# Distribution

- Shipping Python is hard
- Wheels typically have to be build for every OS and every supported Python version

# maturin

- Tool for building and publishing
- Builds wheels for all installed Python versions

Install:

```
python3 -m venv .env
source .env/bin/activate
pip install maturin
```

# maturin Demo

# abi3

Use Python's stable C API to create wheels all modern Python versions can use

```toml
[package]
edition = "2021"
name = "normalizer"
version = "0.1.0"
[lib]
crate-type = ["cdylib"]
name = "normalizer"
[dependencies]
shared = {path = "../shared"}
[dependencies.pyo3]
features = ["extension-module", "abi3-py36"]
version = "0.15.0"
```

# Useful links

- https://github.com/PyO3/pyo3
- https://github.com/PyO3/maturin
- https://github.com/PyO3/setuptools-rust
- Using Python from Rust
- Comparing different methods of accelerating numerical python code

# uniffi-rs

- A multi-language bindings generator for Rust
- Developed by Mozilla for Mozilla
- Targets Swift & Kotlin
- (Python & Ruby are supported to, but not that much talked about)
- License: MIT

# UniFII Diagram

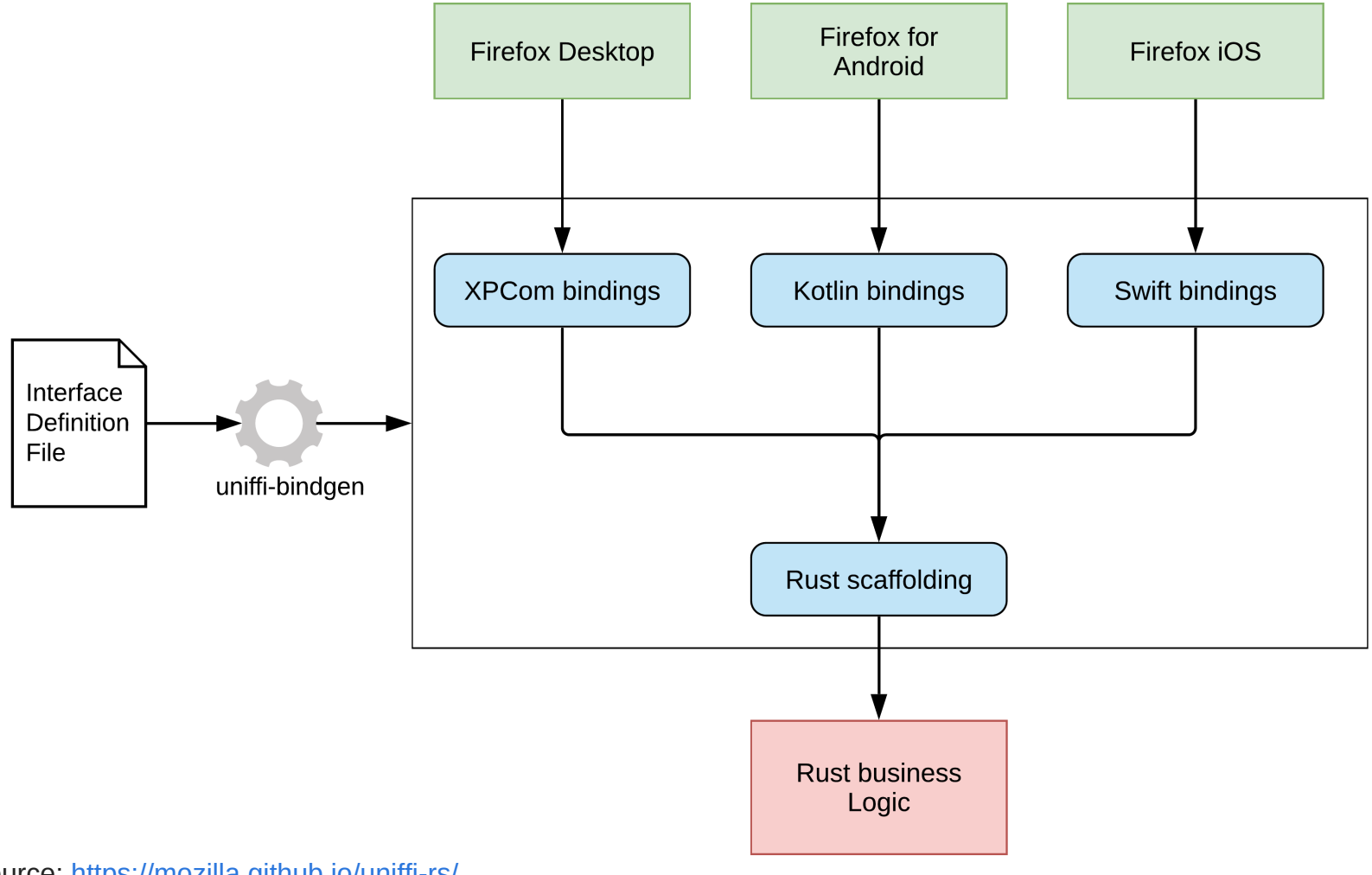



Image Source: https://mozilla.github.io/uniffi-rs/

# UniFII udl

```
1 namespace normalizer {
2     string normalize_username(string s);
3 };
```

Create bindings:

```
uniffi-bindgen generate src/normalizer.udl --language kotlin
# creates ./src/uniffi/normalizer/normalizer.kt

uniffi-bindgen generate src/normalizer.udl --language swift
# creates ./src/normalizer.swift &
#   ./src/normalizerFFI.h &
#   ./src/normalizerFFI.modulemap
```

# flutter_rust_bridge

High-level memory-safe binding generator for Flutter/Dart

Did not work for me :(

# Additional Link Collection

- https://areweextendingyet.github.io/
- Behind the scenes of 1Password for Linux (2021)
- Supercharge Your NodeJS With Rust (2021)
- Rust in Production: 1Password (2021)
- Performance Comparison: Rust vs PyO3 vs Python (2020)
- Building a fast Electron app with Rust (2018)
- https://thlorenz.com/rid-site - Flutter bindings (Sponsorware)
- allo_isolate - Run Multithreaded Rust along with Dart VM
- libsignal-client uses Rust from NodeJS

# Thanks for your Attention