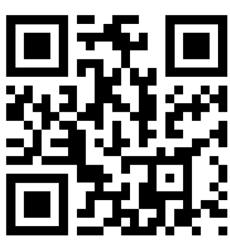
Rust в мобильной разработке

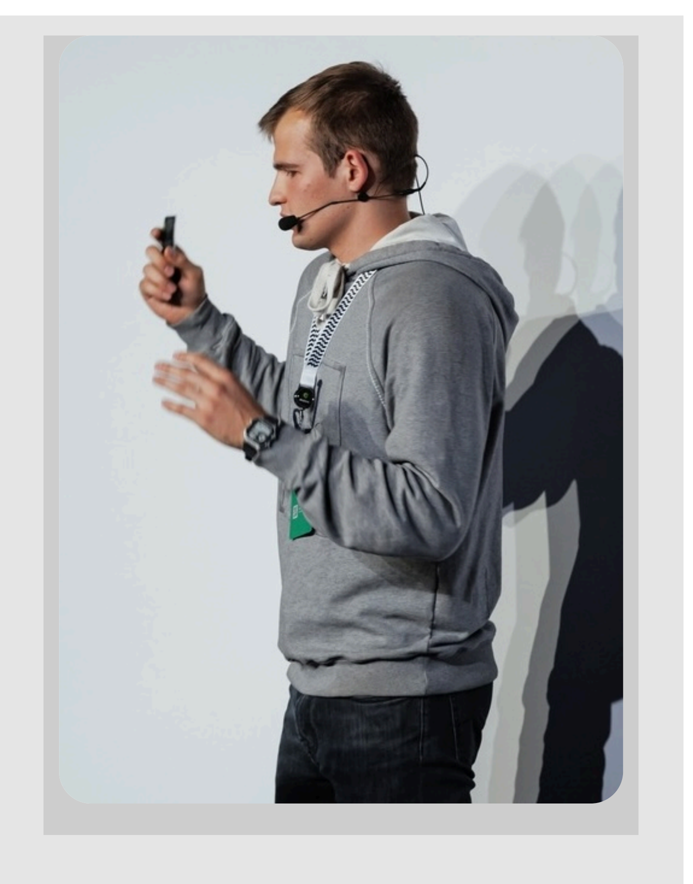
Власюк Александр

Mobius 2026

Власюк Александр

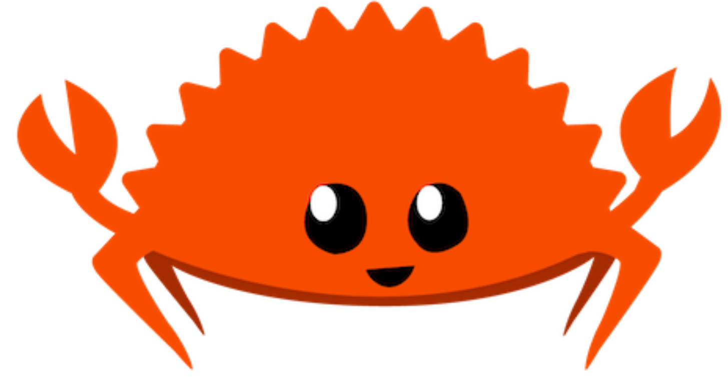
- Android-разработчик в Яндексе
- "Записки Инженера" - @avvlased





План

1. **Введение в Rust**, примеры использования
2. **FFI** – вызов Rust из Swift/Kotlin
3. **UniFFI** – автогенерация биндингов
4. **CRUX** – аналог KMP на Rust
5. **Кроссплатформенный UI** на Rust
6. **Заключение.** Выводы



Зачем нужен Rust?

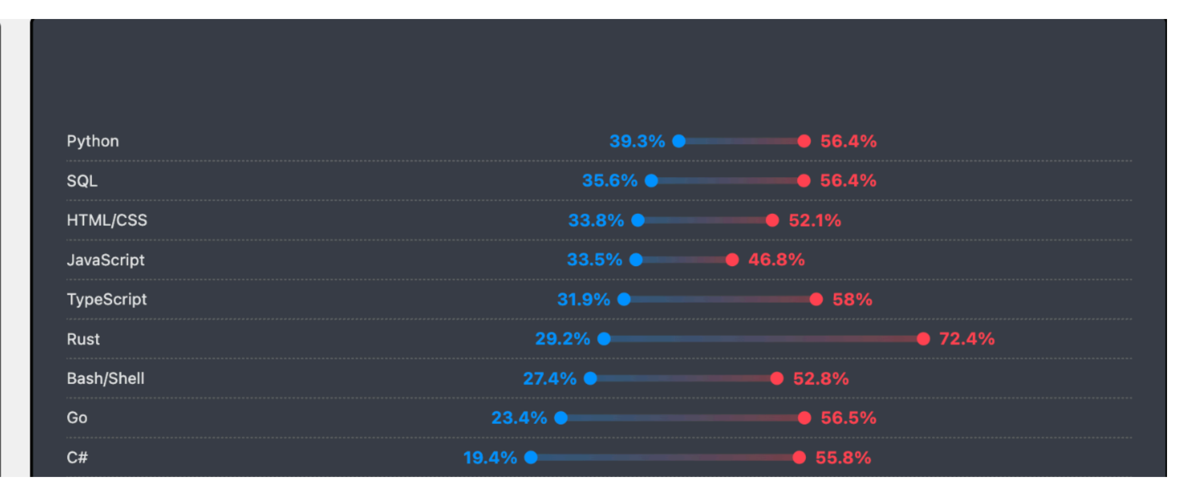
Rust

- **Производительный и безопасный**
- **Современный**. Мультипарадигменный, со строгой типизацией и отличным тулингом.
- **Обеспечивает безопасность памяти на этапе компиляции (без Runtime и GC)**
- **Компилируется в LLVM байткод**
- **"If it compiles, it will work"**

Rust – любимый язык программистов по опросам StackOverflow

Programming, scripting, and markup languages

Rust is yet again the most admired programming language (72%), followed by Gleam (70%), Elixir (66%) and Zig (64%). Gleam is a new addition to the list, and for good reason - developers like it!



Источник: StackOverflow Dev Survey 2025

Rust и AI

- **Rust продолжает набирать популярность** на фоне пузыря ИИ
- **Хорошо подходит для AI agentic кодинга** за счет своей строгости:

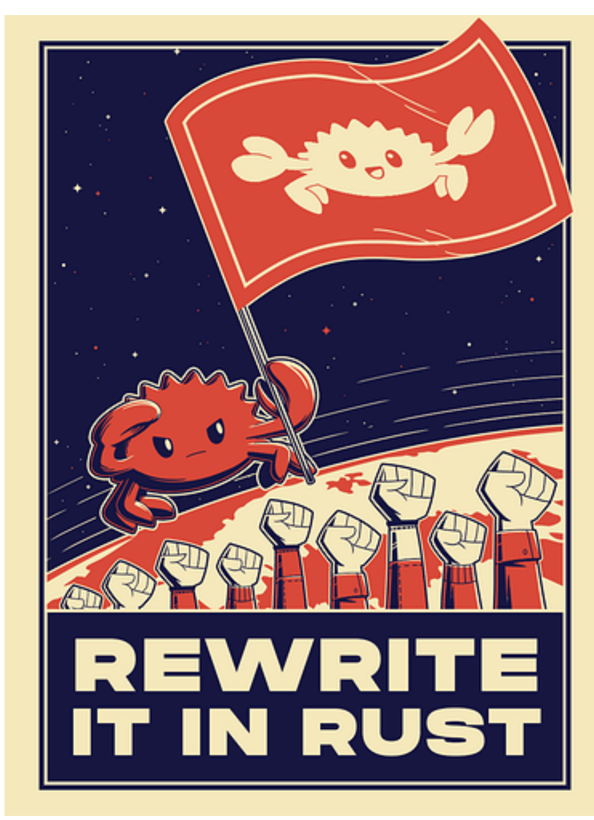
  "На Rust трудно запустить неработающий код"

Windows 11

Microsoft building team to eliminate C and C++, translate code to Rust using AI, as Windows 11 adopts Rust

Источник: Windows Latest

#RIIR



Источник: CaffeinatedCoder @ Medium

Мотивация - Rust в мобильных приложениях

- **Переиспользование логики** между платформами
- **Нативные библиотеки** – криптография, сетевые протоколы, кодеки, парсеры
- **Производительность** – без рантайма и GC
- **Безопасность памяти**

Синтаксис Rust

```rust
// Переменные и типы
let name: &str = "Rust";
let mut count: i32 = 0;

// Enum с данными (как в Swift)
enum NetworkResult {
    Success(String),
    Error { code: i32, message: String },
}

// Pattern matching
match result {
    NetworkResult::Success(data) => {
        println!("Данные: {}", data);
    }
    NetworkResult::Error { code, .. } => {
        println!("Ошибка: {}", code);
    }
}
```

Traits и Generics

```rust
// Trait ≈ Protocol (Swift) / Interface (Kotlin)
trait ApiService {
    fn fetch(&self, url: &str) -> Result<String, Error>;
}

struct HttpClient;

impl ApiService for HttpClient {
    fn fetch(&self, url: &str) -> Result<String, Error> {
        Ok("response".to_string())
    }
}

fn load<T: ApiService>(service: &T) {
    let result = service.fetch("/api/data");
}
```

Ownership

У каждого значения ровно один владелец.

```
let data = "hello";           // `data` владеет строкой
let data2 = data;             // владение перешло к `data2`

println!("{:?}", data);       // ❌ ОШИБКА КОМПИЛЯЦИИ!
                               // `data` больше не владеет данными

println!("{:?}", data2);      // ✅ OK — `data2` владелец
```

Компилятор точно знает, когда нужно освободить память – без GC и ручного управления

Borrow Checker

Можно не передавать владение, а **заимствовать** (`&`)

```rust
fn print_len(data: &String) { // заимствование
    println!("Длина: {}", data.len());
} // заимствование заканчивается

fn add_text(data: &mut String) { // мутабельное заимствование
    data.push_str("!");
}

// Правило: сколько угодно &, ИЛИ одна &mut
let mut text = String::from("Rust");
print_len(&text);            // ✅ иммутабельное заимствование
add_text(&mut text);         // ✅ мутабельное заимствование
print_len(&text);            // ✅ снова можно читать
```

Borrow checker **гарантирует** отсутствие data races на этапе компиляции.

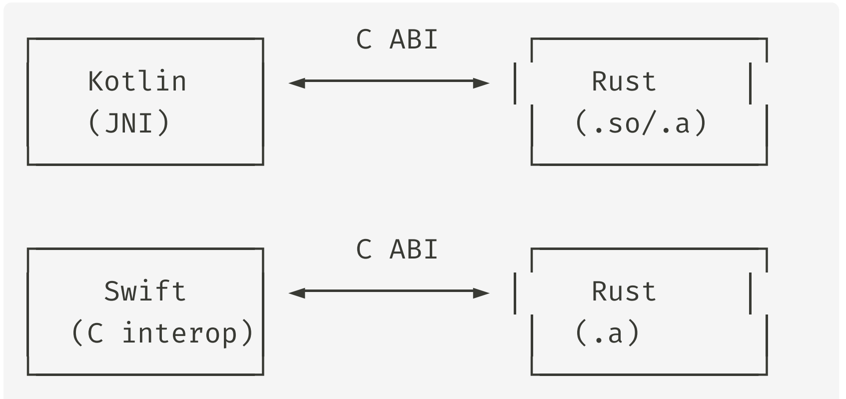
Rust vs Swift/Kotlin/C++

	Swift	Kotlin	C++	Rust
Modern Language	✅	✅	⚠️	✅
Tooling	✅ SPM/Cocoapods	✅ Gradle	⚠️ Wtf?	✅ Cargo
Mobile	✅ Native	✅ Native	⚠️ JNI/bridging	✅ (Uni)FFI
Memory Safety	⚠️ Runtime	⚠️ Runtime	⚠️ None	✅ Compile-Time
Performance	⚠️	⚠️	✅	✅

FFI

Что такое FFI?

- **Foreign Function Interface** – механизм вызова функций между языками
- Rust компилируется в **нативный код** – можно вызывать через C ABI
- Два направления:
  - Rust → вызывает C/C++ код
  - **C/Swift/Kotlin → вызывают Rust код**



Ручной FFI в Rust

```rust
#[no_mangle]
pub extern "C" fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

- `#[no_mangle]` – сохраняет имя функции
- `extern "C"` – используем C ABI (calling convention)

Вызов из Swift (iOS)

1. Bridging Header

```c
// RustLib.h
int32_t add(int32_t a, int32_t b);
```

2. Вызов в Swift

```swift
// ViewController.swift
import Foundation

let result = add(2, 3)
print("2 + 3 = \(result)") // 5
```

Сборка:

- `cargo build --target aarch64-apple-ios` → `libmylib.a`
- Подключить `libmylib.a` + `RustLib.h` в Xcode

Вызов из Kotlin (Android)

1. JNI обёртка в Rust

```rust
use jni::JNIEnv;
use jni::objects::JClass;

#[no_mangle]
pub extern "system" fn
  Java_com_example_RustLib_add(
    _env: JNIEnv,
    _class: JClass,
    a: i32,
    b: i32,
) -> i32 {
    a + b
}
```

2. Вызов в Kotlin

```kotlin
// RustLib.kt
class RustLib {
    companion object {
        init {
            System.loadLibrary("mylib")
        }

        external fun add(a: Int, b: Int): Int
    }
}

// Использование
val result = RustLib.add(2, 3)
println("2 + 3 = $result") // 5
```

Сборка:

- `cargo build --target aarch64-linux-android` → `libmylib.so`
- Положить `.so` в `jniLibs/arm64-v8a/`

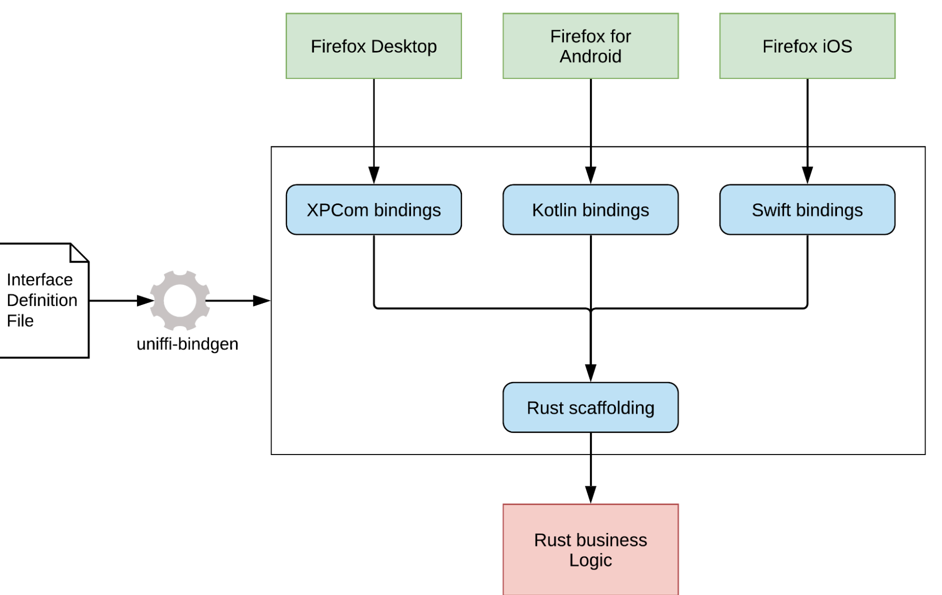
Проблемы ручного FFI

- C ABI не поддерживает сложные типы (String, Vec, enum)
- Boilerplate – JNI-обёртки, хедеры .h
- Ручное управление памятью через границу
- Легко допустить ошибку (утечки памяти, UB)

Решение → UniFFI

UniFFI

- Проект **Mozilla**
- Автоматическая генерация FFI биндингов в Rust



Поддерживаемые языки

Official:

- Swift
- Kotlin/JVM
- Python
- Ruby

Third Party:

- C
- C++
- C#
- Java
- Dart
- Go
- Kotlin Native

UniFFI – простой пример

```rust
// lib.rs

#[uniffi::export]
fn hello(name: String) -> String {
    format!("Hello, {}!", name)
}

uniffi::setup_scaffolding!();
```

UniFFI - интеграция в проект

iOS (Xcode):

1. `cargo run --bin uniffi-bindgen generate ...` → `.swift` файл
2. Добавить `.a` + `.swift` в Xcode проект

Android (Gradle):

- **Rust Android Gradle Plugin**
- **Gobley** (для KMP проекта)

UniFFI – сгенерированный Swift

```swift
// uniffi-bindgen generated
public func hello(name: String) -> String { ... }
```

Вызов в Swift:

```swift
import MyRustLib

let greeting = hello(name: "World")  // "Hello, world!"
```

Типы конвертируются автоматически: `String` ↔ `String`, `Vec<T>` ↔ `[T]`

UniFFI – сгенерированный Kotlin

```
// uniffi-bindgen generated
fun hello(name: String): String { ... }
```

Вызов в Kotlin:

```
import com.example.mylib.*

val greeting = hello("World")     // "Hello, world!"
```

Работает через JNA (Java Native Access), без ручного JNI

UniFFI — async функции

```rust
#[uniffi::export]
async fn fetch_data(url: String) -> Result<String, AppError> { ... }
```

Swift:

```swift
let data = try await fetchData(url: "https://...")
```

Kotlin:

```kotlin
val data = fetchData("https://...")  // suspend fu
```

`async fn` → Swift `async/await`, Kotlin `suspend fun`

UniFFI — callback-интерфейсы

Вызов платформенного кода из Rust:

```
#[uniffi::export(callback_interface)]
pub trait Logger {
    fn log(&self, level: String, message: String);
}

#[uniffi::export]
fn do_work(logger: Box<dyn Logger>) { ... }
```

```
// Swift
class SwiftLogger: Logger {
    func log(level: String, message: String) {
        print("[\(level)] \(message)")
    }
}
doWork(logger: SwiftLogger())
```

UniFFI – custom types

Маппинг кастомных типов на нативные типы платформ:

```
# Rust UtcDateTime → Kotlin java.util.Date
[bindings.kotlin.custom_types.UtcDateTime]
type_name = "java.util.Date"
into_custom = "java.util.Date({})"
from_custom = "{}.time"
```

UniFFI – выводы

Плюсы:

- Простая интеграция – минимум boilerplate
- Поддержка async/await из коробки
- Callback-интерфейсы для вызова платформенного кода
- Поддержка объектов, enum, ошибок

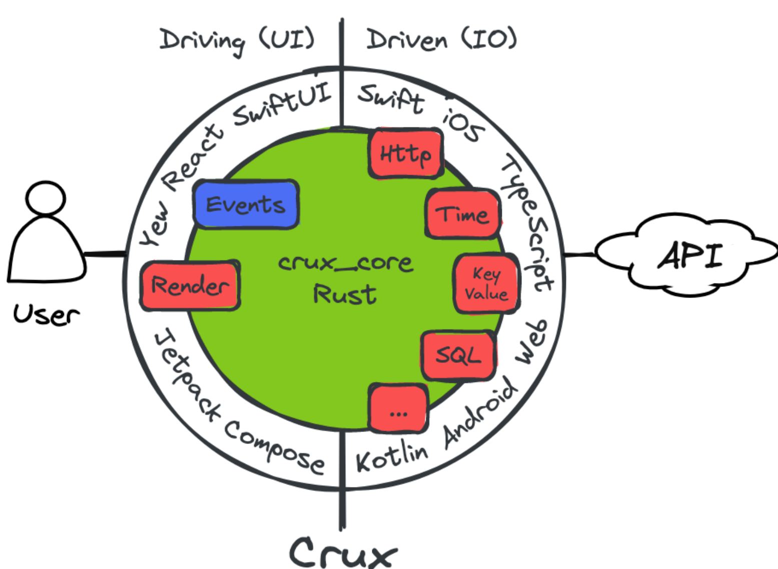
Минусы:

- Нет generics – нужно конкретизировать типы
- Нет интеграции с GC JVM – нужен `.destroy()` / `.use {}`
- Overhead от сериализации через FFI-границу
- Ручная поддержка для сложных типов данных

Crux - KMP на Rust

Crux

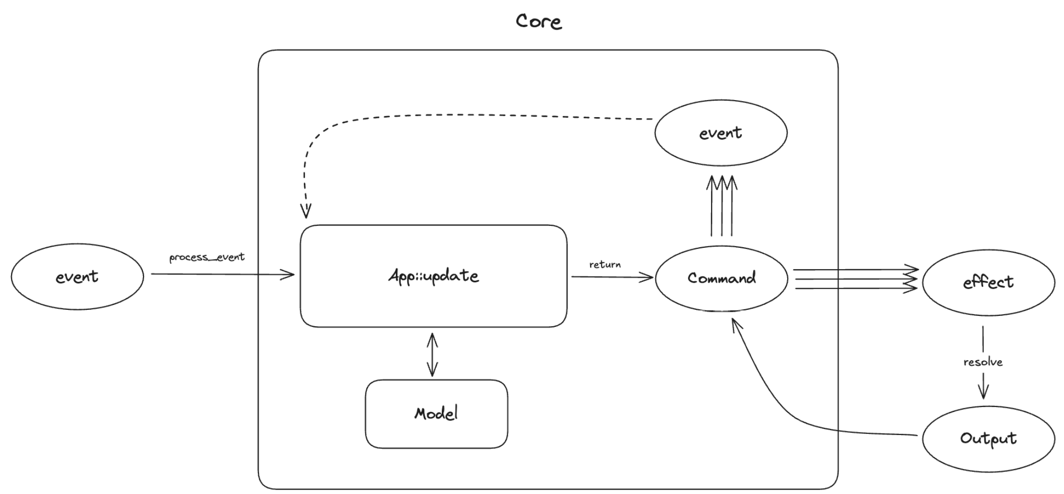
- Архитектурный фреймворк + набор инструментов
- Вся бизнес-логика – в Rust **Core**
- UI – нативный: SwiftUI, Compose, React



Crux – Core + Shell

Архитектура вдохновлена **TEA** (The Elm Architecture):

- **Core** (Rust) – чистый, без сайд-эффектов
  - `Model` – состояние
  - `Event` – входные события
  - `update()` – логика переходов
  - `ViewModel` – данные для UI
- **Shell** (Swift/Kotlin/TS) – тонкий слой
  - Рисует UI
  - Выполняет эффекты (HTTP, KV, …)
  - Передаёт события в Core



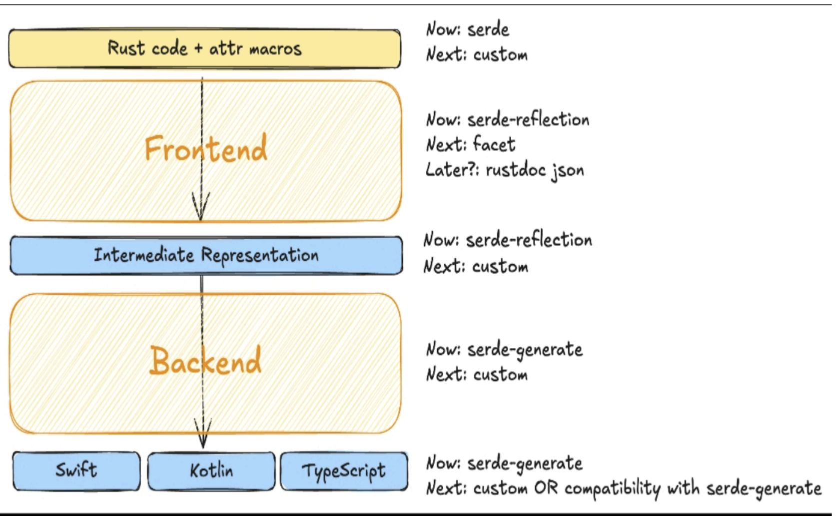
Crux – биндинги и typegen

Двойной подход к FFI:

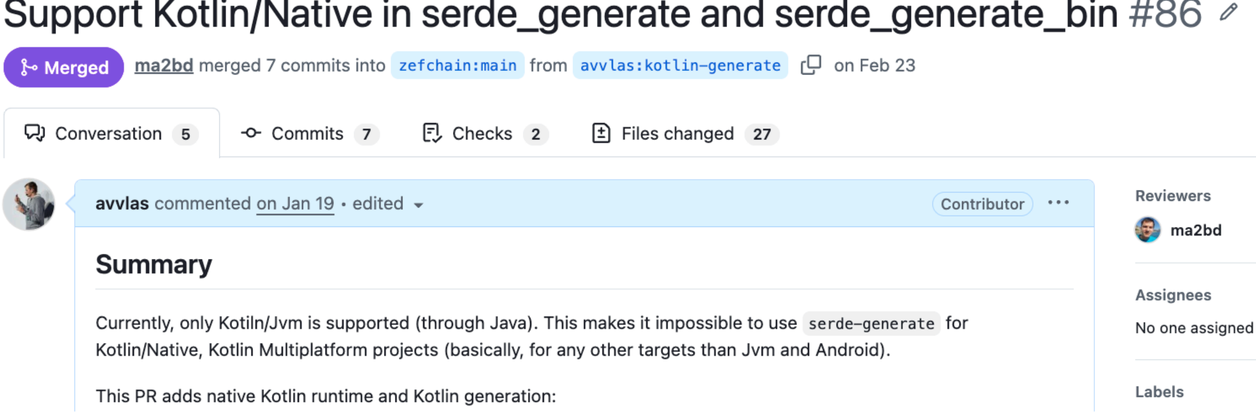
1. **UniFFI** – механизм вызова функций через FFI
2. **Кастомный typegen (serde/facet)** – генерация типов данных

Почему не хватает одного UniFFI?

- UniFFI не поддерживает сложные типы данных
- UniFFI не поддерживает Web
- Нужен единый подход для Swift + Kotlin + TypeScript

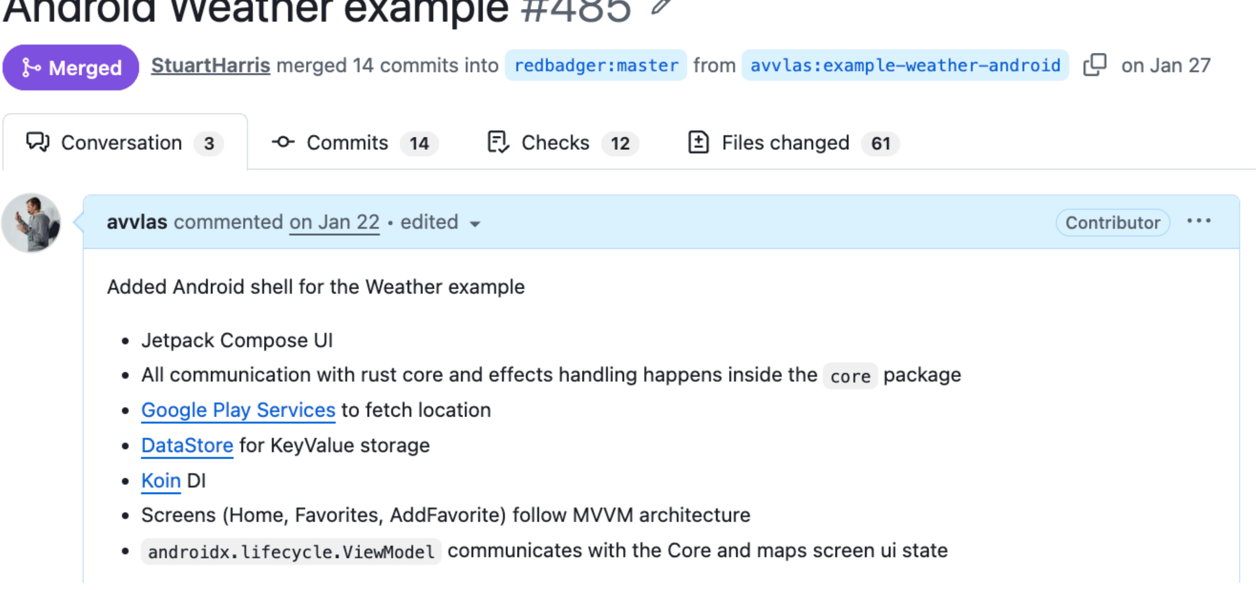


Crux – поддержка Kotlin/Native



Facet-generate pull #48 · Serde-reflection pull #86

Crux – Android пример



Crux pull #485

Crux в Proton

- Набор open-source утилит-приложений
- Заадоптили CRUX, рассказали на RustNation: https://github.com/ProtonMail/proton-rust-nation-2026

Источник: Proton

Crux – текущий стейт

Плюсы:

- Активно развивается
- Отзывчивое сообщество
- Тестируемость core-логики
- Кастомный typegen
- Поддерживает web (wasm-bindgen)

Минусы:

- Pre-1.0, API может меняться
- Бойлерплейт в инфраструктурном/bridge коде
- Мало примеров real-world приложений*

Кроссплатформенный UI на Rust

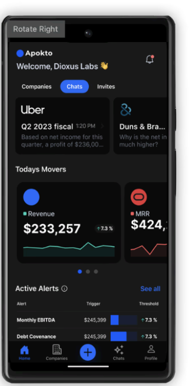
Dioxus

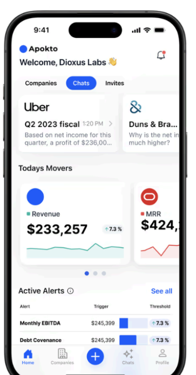
- **React-like** фреймворк на Rust
- Web, Desktop, iOS, Android
- CLI `dx` – serve, bundle, hot-reload

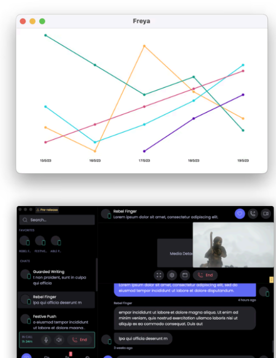
One codebase, every platform.

Dioxus is *the* Rust framework for building fullstack web, desktop, and mobile apps.

Iterate with live hotreloading, add server functions, and deploy in record time.







Dioxus – Пример

```rust
use dioxus::prelude::*;

fn main() {
    dioxus::launch(App);
}

#[component]
fn App() -> Element {
    let mut count = use_signal(|| 0);
    rsx! {
        div { "Счётчик: {count}" }
        button { onclick: move |_| count += 1, "+
        button { onclick: move |_| count -= 1, "-
    }
}
```

Tooling:

```
# Создать проект
dx new

# Запуск на iOS-симуляторе
dx serve --platform ios

# Запуск на Android
dx serve --platform android

# Продакшн-сборка
dx bundle
```

Dioxus - Рендеринг

Текущий подход (0.6):

- На мобилках рендерится в **WebView**
- iOS – WKWebView, Android – Android WebView
- VirtualDom → HTML/CSS через JS-бридж

Dioxus Native (0.7):

- Кастомный рендерер на WGPU (Blitz)
- Нативно, без WebView

Slint

- **Декларативный GUI-тулкит** для Rust
- Свой DSL `.slint`
- **Можно выбрать реднеринг**:
  - QT
  - Skia
  - FemtoVG
  - Software renderer (CPU, для embedded)

```
// counter.slint
export component Counter {
    in-out property <int> count: 0;

    VerticalLayout {
        Text { text: "Count: \{count}"; }
        Button {
            text: "+";
            clicked => { count += 1; }
        }
    }
}
```

Makepad

- **100% GPU-рендеринг** – Metal, DirectX, OpenGL, WebGL
- Live-design DSL с hot-reload
- iOS и Android – через `cargo-makepad`
- Makepad 1.0 вышел в 2025

Robius

- **Мета-проект** для мобильной разработки на Rust
- **Makepad** для UI
- **Osiris** – абстракции над платформенными API (камера, GPS, уведомления, хранилище, сеть)
- **Цель** – не писать платформенный код
- **Robrix** – Matrix-клиент на чистом Rust (Makepad + Robius)

Сравнение фреймворков

	Dioxus	Slint	Makepad
Парадигма	React-like, RSX, signals	Декларативный DSL	Hybrid retained/immediate
Рендеринг	WebView (0.6) / WGPU (0.7)	Custom (Skia/FemtoVG/CPU)	100% GPU
Hot reload	Да	Да	Да
Accessibility	Через WebView	Да (Narrator)	Нет
Зрелость	Pre-1.0, быстро развивается	1.15, стабильный	1.0

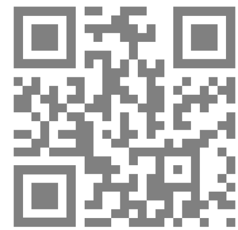
Rust GUI для мобилок – текущий стейт

- Ни один фреймворк пока не достиг зрелости Flutter / React Native
- Dioxus – самый удобный developer experience
- Robius – амбициозная попытка решить проблему платформенных API

Выводы

- **Rust – зрелый инструмент** для shared-логики в приложениях (Mozilla, 1Password, Signal, Matrix)
- **FFI** – фундамент интеграции
- **UniFFI** – генерация биндингов из коробки, убирает бойлерплейт
- **Crux** – архитектура и тулинг поверх UniFFI: shared state, эффекты, тестируемость, Web через WASM
- **Dioxus / Slint / Makepad** – Rust UI пока не зрелый, но быстро развивается

Спасибо!



Власюк Александр

Mobius 2026