



# Ein Core, sie zu rosten

Langlebige & flexible Cross-Platform-Applikationen

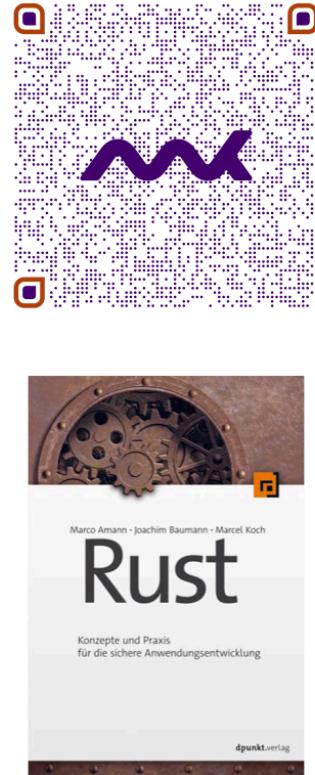


# Nahbare Softwareberatung

Team von 7  
Web  
Gewaltfreie  
Kommunikation

Mobile Apps  
Trainings  
Methoden

[www.marcelkoch.net](http://www.marcelkoch.net)



## Warum Cross-Platform?

- Webtechnologien dominieren
- Native Apps weiterhin wichtig (z.B. Hardwarezugriffe, Performance)

Native Apps bieten...

Vorteile

Performance

Nachteile

spezifische Code Base

Look & Feel

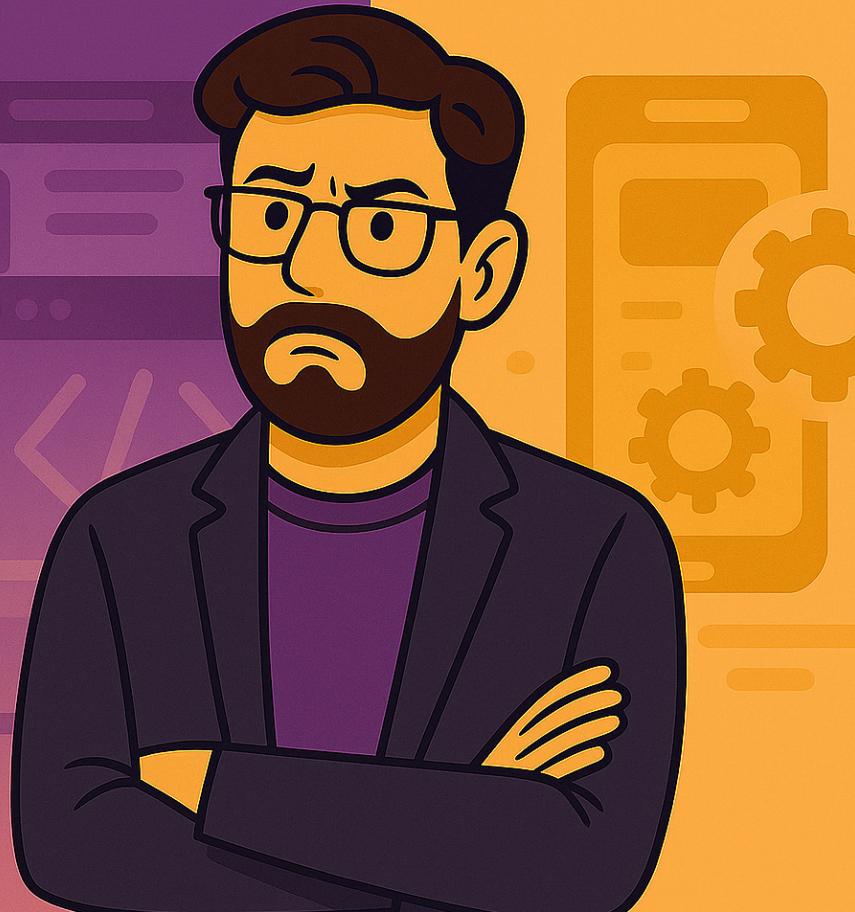
unterschiedliche Konzepte / APIs

Hardwarezugriff

umständliche Installation

WEB-  
TECHNOLOGIEN

NATIVE  
ANWENDUNGEN



# Cross-platform frameworks

	Flutter	React Native	Xamarin / MAUI	Kotlin Multiplatform	Qt
Language	Dart	JavaScript / TypeScript	C#	Kotlin	C++
Company	Google	Facebook	Microsoft	Jetbrains	The Qt company

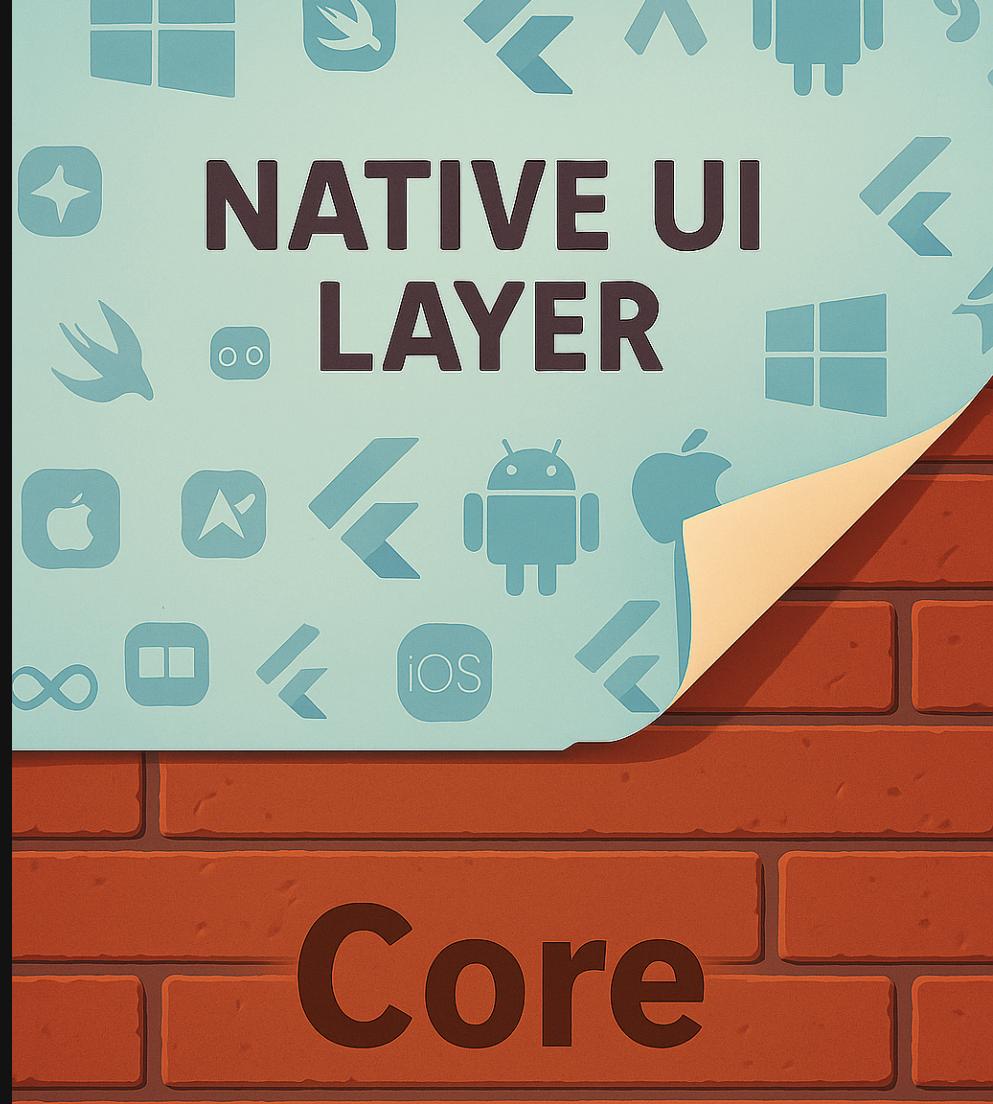


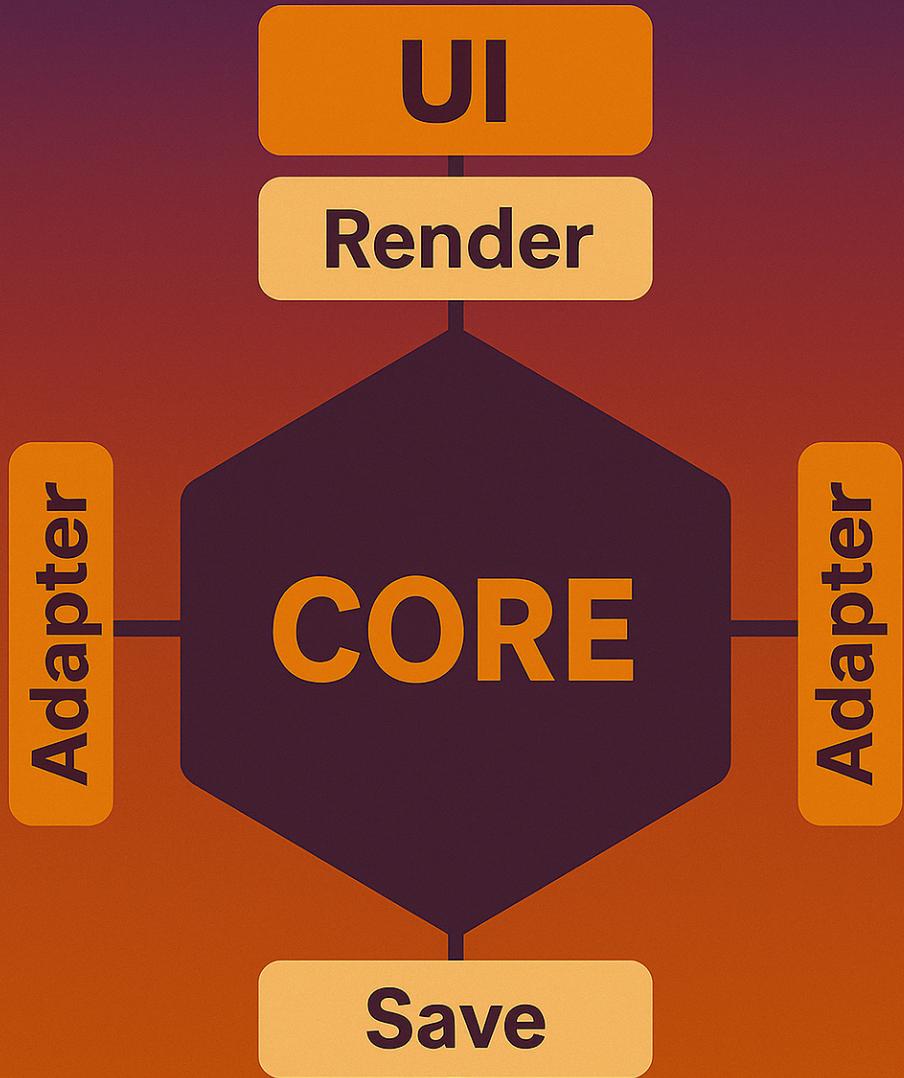
## Gut für schnelle Ergebnisse

- Schnelle Ergebnisse durch (oft dynamische) Sprachen
- Gut für kurzlebige Apps (1–2 Jahre)
- Für langlebige Projekte (10+ Jahre) kritischer zu betrachten:
  - Wird das Framework langfristig gepflegt?
  - Reicht Support für Android/iOS/Web?
  - Genügt Performance einer WebView?
  - Wie gut ist die Testbarkeit?
  - Finde ich langfristig Entwickler?

# Langlebigkeit durch separaten Core

- UI-Technologie wandelt sich schnell
- Idee: Trennung von Core & UI (z.B. hexagonale Architektur)
- Core enthält Geschäftslogik und stabile Bestandteile
- UI und Plattform-APIs als externe Ports
- Beispiele für Ports: Kamera, Dateisystem, UI



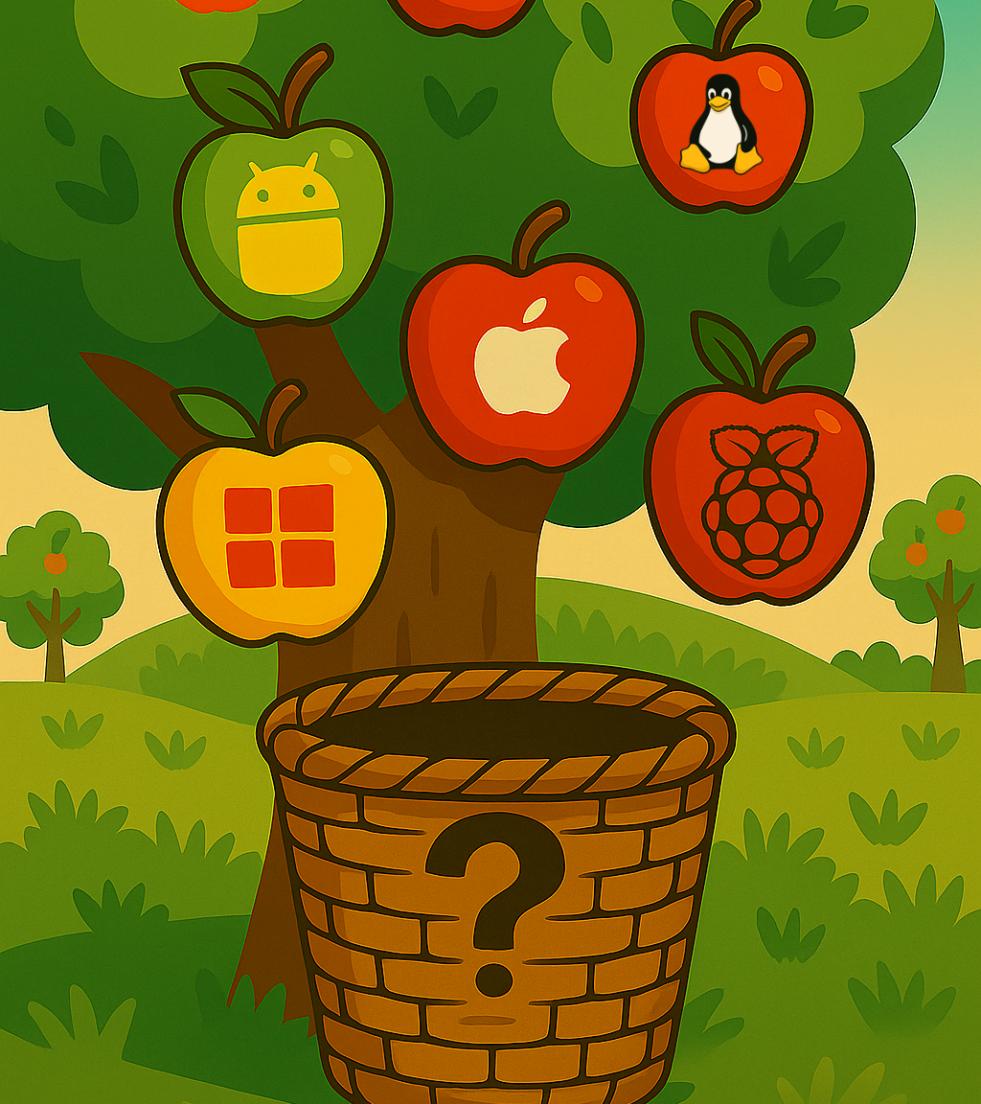


## Ports and Adapters

- Render: Port
- UI: Adapter

# Wahl der Core-Sprache

- Zielplattformen: Android, iOS, Windows, macOS, Raspberry Pi
- Wichtige Kriterien:
  - Stabilität (wenig grundlegende Änderungen)
  - Robustheit (gut wartbarer Code)
  - Langlebigkeit (Unterstützung durch große Firmen)
  - Flexibilität (Einsetzbarkeit auf vielen Plattformen inkl. Web)



# Rust für den Core

- Wartbarer, performanter Core mit Rust
- Plattformunterstützung: Desktop, Mobile, Web (via WASM), SBCs
- Explizite Syntax und Compiler unterstützen langfristige Wartbarkeit
- Lernkurve höher, kleinere Community als JS oder C
- Wachsende Community, viele Migrationen von C nach Rust
- Unterstützung durch Firmen wie AWS, Google, Meta, Microsoft



# Architektur des Cores

- Mehr Logik im Core = mehr Wiederverwendbarkeit
- UI wird schlanker und reagiert auf ViewModel

# UI-Aktionen als fachliche Aktionen (Domain Events)

- UI-Events wie `onClick` werden in fachliche Aktionen übersetzt
- Aktionen sind die Eingangsschnittstelle in den Core
- Beispiel in Rust (Speichern von E-Mail und Name):

```
pub enum Actions {  
    ChangeName(String),  
    ChangeEMail(String),  
    ApplyChanges,  
}
```

# Zustand im Core

- Core verwaltet den Zustand, nicht die UI
- Beispiel:

```
struct State {  
    name: String,  
    email: String  
}
```

# Implementierung des Core

- Zentrale Struct `App` verarbeitet Aktionen und liefert ViewModel
- Trennung von State, Actions und ViewModel

```
impl Core {  
    pub fn new() -> Core {  
        Core {  
            state: State::default()  
        }  
    }  
  
    pub fn do_action(&mut self, action: Actions) -> ViewModel {  
        match action {  
            Actions::ChangeName(name) => {  
                self.state.name = name;  
            }  
            Actions::ChangeEMail(email) => {  
                self.state.email = email;  
            }  
            Actions::ApplyChanges => {}  
        }  
        self.render_view_model()  
    }  
}
```

```
pub fn render_view_model(&self) -> ViewModel {  
    ViewModel{  
        name: self.state.name.clone(),  
        email: self.state.email.clone()  
    }  
}  
  
impl Default for Core {  
    fn default() -> Self {  
        Self::new()  
    }  
}
```

# Validierung im Core (1)

- ViewModel enthält auch Fehlermeldungen
- E-Mail-Validierung im Core
- Beispiel: Prüfung auf @

```
enum EMailErrors {  
    NoAt,  
}  
  
struct StateField<F, E> {  
    field: F,  
    error: Option<E>,  
}  
  
struct State {  
    name: String,  
    email: StateField<String, EMailErrors>,  
}
```

```
pub struct ViewModelField<F> {  
    value: F,  
    error: Option<String>,  
}  
  
pub struct ViewModel {  
    pub name: String,  
    pub email: ViewModelField<String>,  
}
```

# Validierung im Core (2)

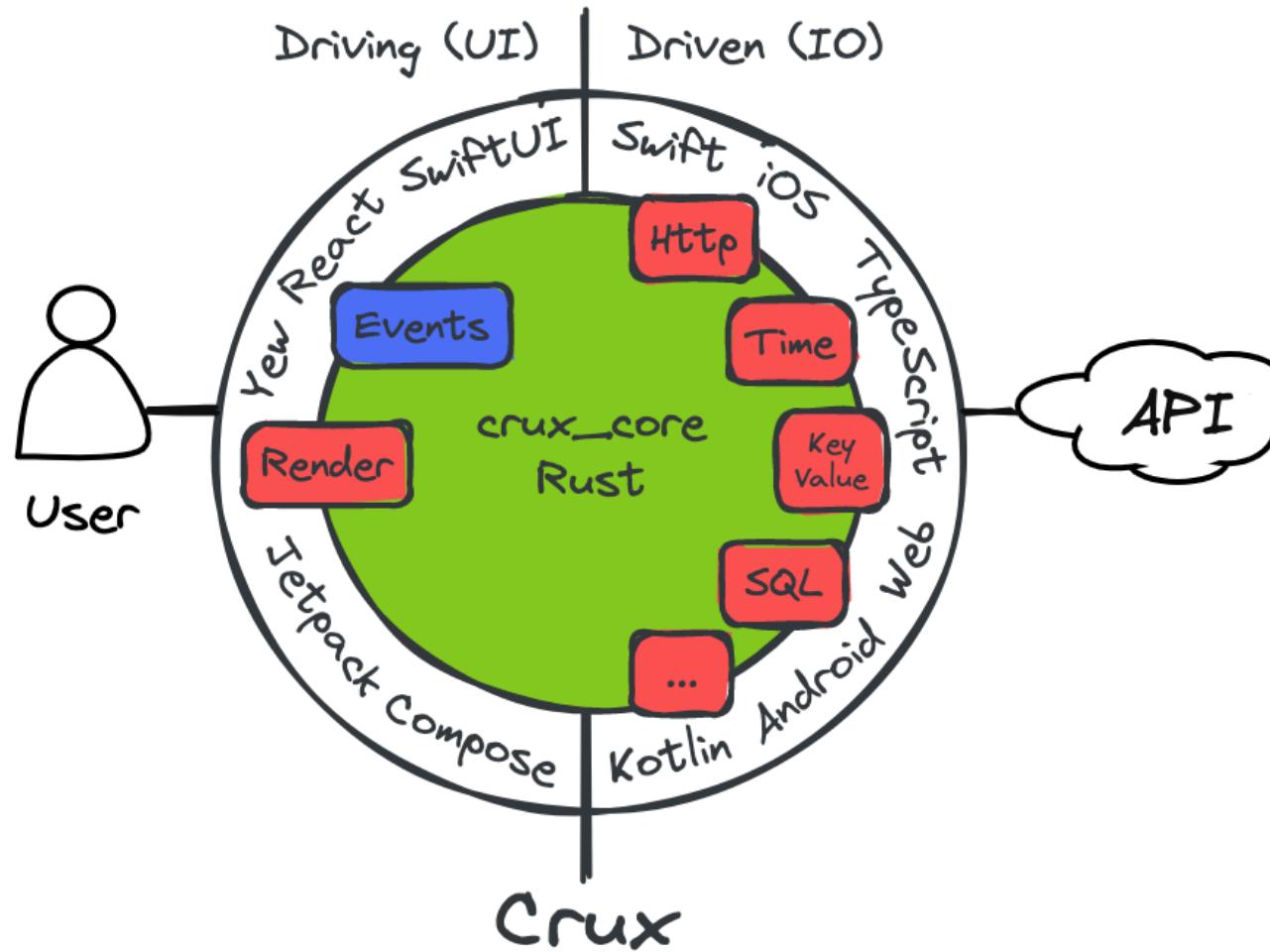
- In `do_action()` und `render_view_model()`

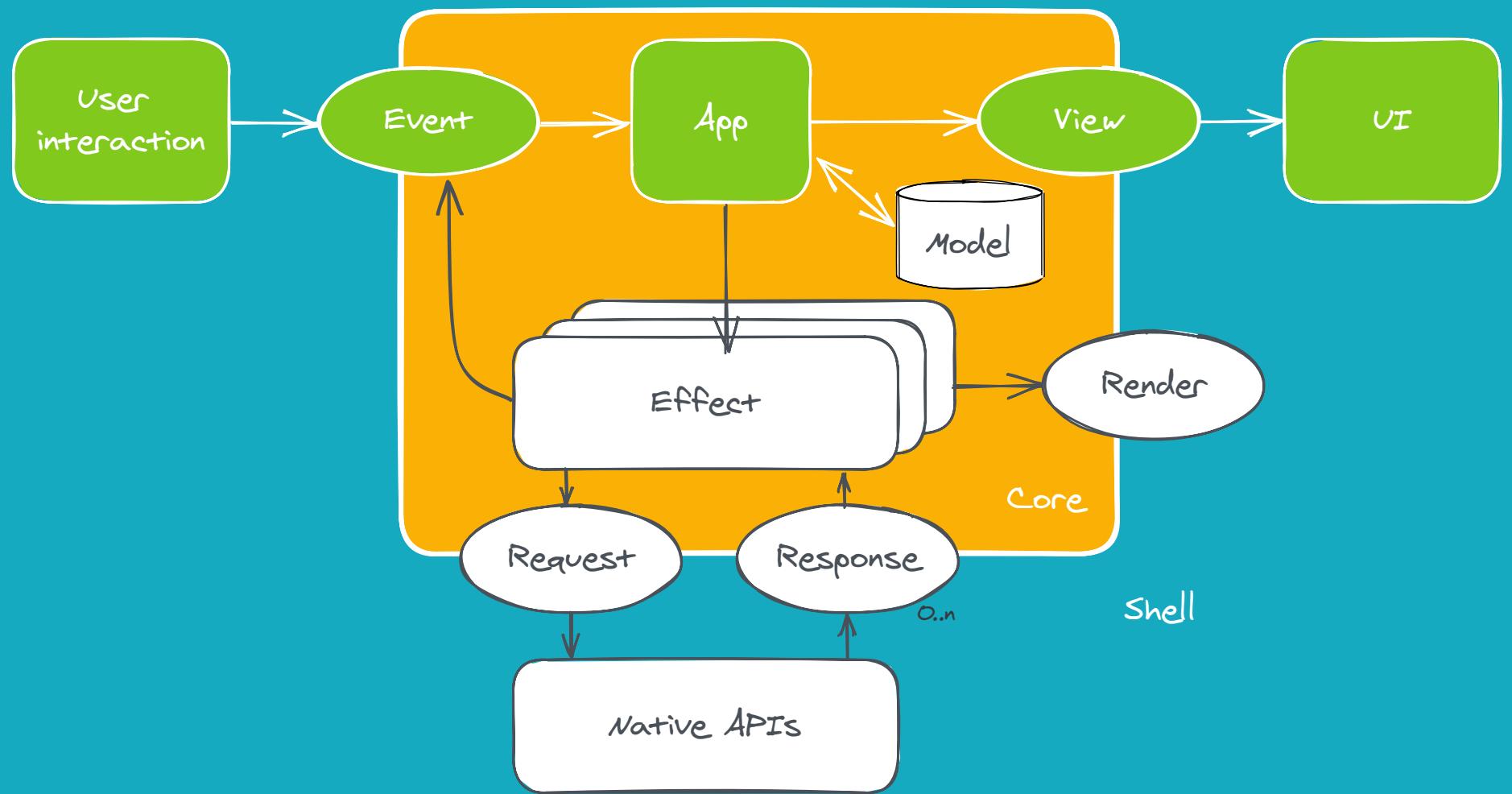
```
Actions::ChangeEmail(email) => {
    let email_error = if email.contains("@") {
        None
    } else {
        Some(NoAt)
    };
    self.state.email = StateField {
        field: email,
        error: email_error,
    };
}
```

```
ViewModel {
    name: self.state.name.clone().into(),
    email: ViewModelField {
        value: self.state.email.field.clone().into(),
        error: match &self.state.email.error {
            None => None,
            Some(_err) => Some("Keine valide E-Mail-Adresse".into()),
        },
    },
}
```

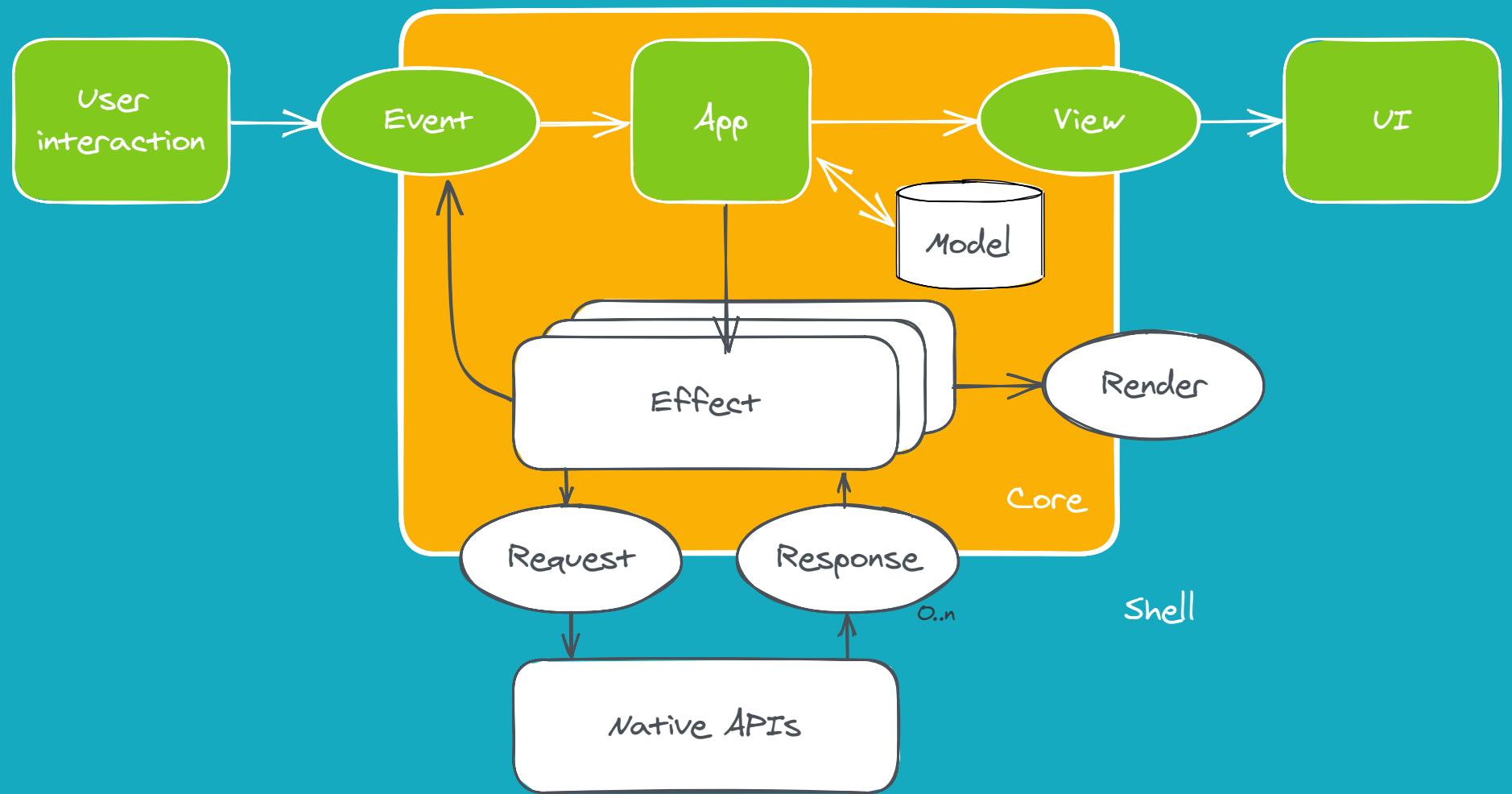
# Crux

- Actions sind Events
- State ist Model
- App
- Effect (Was raus geht)
- Command (Erst Effect, dann Event)
- Operation (Request-Payload)









# Core

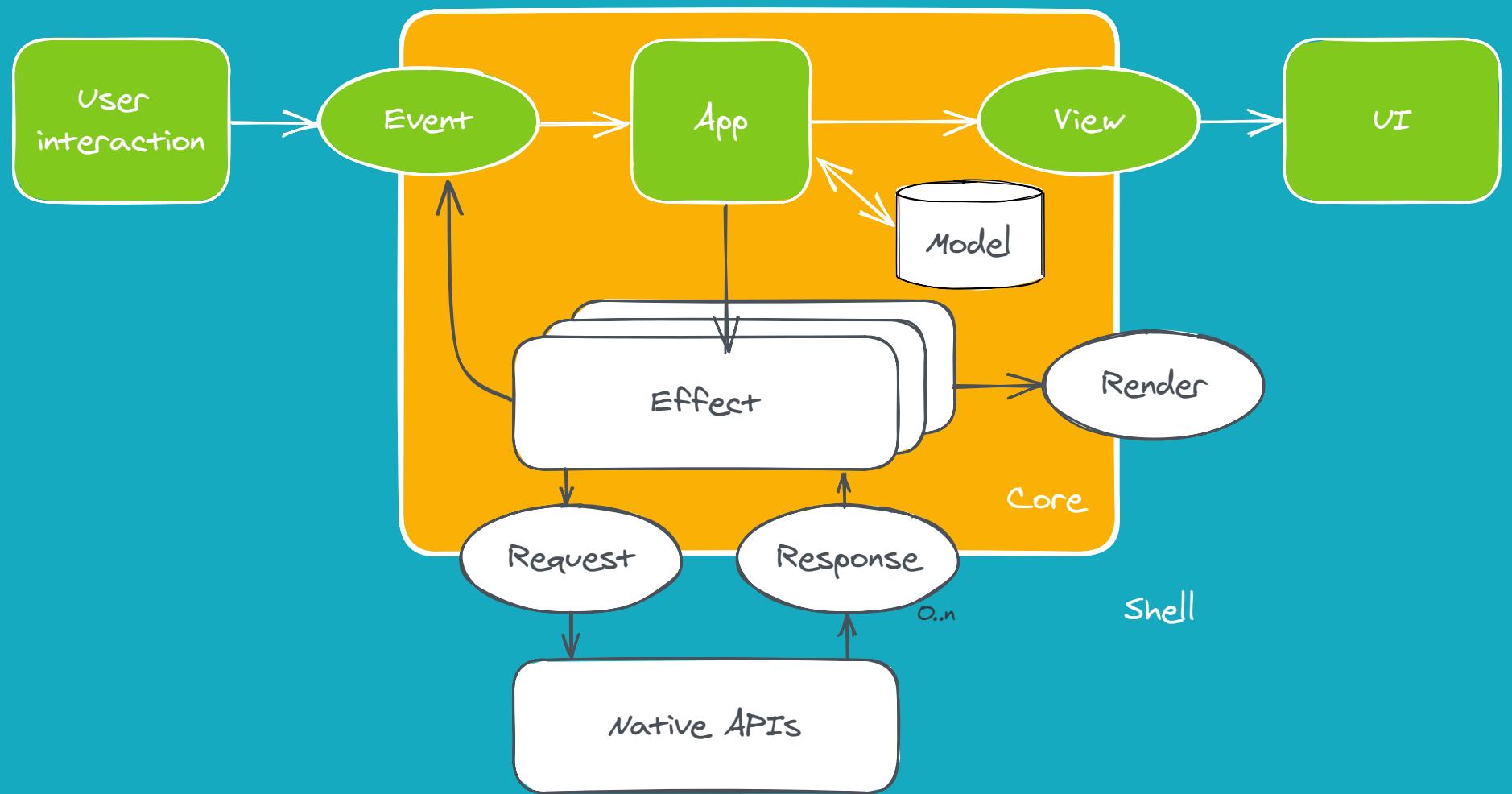
- Umschließt die App
- Hält das Model versteckt

```
let core: Arc<Core<EmailApp>> = Arc::new(Core::new());  
  
let effects: Vec<Effect> =  
    core.process_event(ChangeEmail("m@rcelko.ch".into()));  
  
for effect in effects {  
    match effect {  
        Effect::Render(_) => {  
            let view_model = core.view();  
  
            assert_eq!(view_model.email, "m@rcelko.ch")  
        }  
    }  
}
```

# Bridge

- Umschließt den Core
- Kümmert sich um (De)serialisierung

```
let serialized =  
    bincode::serialize(&ChangeEmail("m@rcelko.ch".into())).unwrap();  
  
let effects: Vec<u8> = bridge.process_event(&serialized).unwrap();  
  
let effects: Vec<Request<EffectFfi>> =  
    bincode::deserialize(effects.as_slice()).unwrap();  
  
for request in effects {  
    let effect = request.effect;  
  
    match effect {  
        EffectFfi::Render(_) => {  
            let view_model = bridge.view().unwrap();  
            let view_model: ViewModel =  
                bincode::deserialize(&view_model).unwrap();  
  
            assert_eq!(view_model.email, "m@rcelko.ch")  
        }  
    }  
}
```



```
1 // ...
2
3 #[effect]
4 pub enum Effect {
5     Render(RenderOperation),
6     StartWritingEmail(StartWritingEmailOp),
7 }
8
9 #[derive(Deserialize, Serialize, Clone, PartialEq, Debug)]
10 pub struct StartWritingEmailOp {
11     pub subject: String,
12 }
13
14 impl Operation for StartWritingEmailOp {
15     type Output = bool;
16 }
```

```
1 impl App for EmailApp {
2     fn update(...) -> Command<Self::Effect, Self::Event> {
3         match event {
4             Event::ChangeEmail(email) => {
5                 model.email = email.clone();
6
7                 let command: Command<Self::Effect, Self::Event> =
8                     Command::request_from_shell(StartWritingEmailOp {
9                         subject: "Moin!".into(),
10                        })
11
12                         .then_send(|output| Event::EmailSent(output));
13                         Command::all([command, render()])
14
15             Event::EmailSent(sent) => {
16                 model.sent = sent;
17                 render()
18             }
19         }
20     }
21     fn view(&self, model: &Self::Model) -> Self::ViewModel {
22         ViewModel {
23             email: model.email.clone(),
24             sent_message: if model.sent {
25                 Some("Erfolgreich geschickt".into())
26             } else {
27                 None
28             },
29         }
30     }
31 }
```

# Tests

- Die App direkt testen
- Den Core innerhalb eines Integrationstests testen

```
#[test]
fn simple_test() {
    let app = EmailApp;

    let mut model = Model {
        email: "hello@example.com".to_string(),
        sent: false,
    };

    let event = Event::EmailSent(true);
    let mut cmd = app.update(event, &mut model, &());
    assert_effect!(cmd, Effect::Render(_));

    let view = app.view(&model);

    assert_eq!(view.sent_message.unwrap(),
               "Erfolgreich geschickt".to_string());
}
```

# Anbindung an die Shell

Kleber für die Tapete

- uniffi (& serde\_generate)
- wasm-pack
- Flutter Rust Bridge
- FFI



# uniFFI

- Mit Macros (oder interface definition file)  
Funktionsschnittstellen in Fremdsprachen generieren

```
static CORE: LazyLock<Bridge<EmailApp>>
= LazyLock::new(|| Bridge::new(Core::new()));

#[uniffi::export]
#[must_use]
pub fn process_event(data: &[u8]) -> Vec<u8> {
    match CORE.process_event(data) {
        Ok(effects) => effects,
        Err(e) => panic!("{}"),  

    }
}
```

# serde\_generate

- Generierte komplexe Typen in Fremdsprachen

```
use serde_generate::{
    Tracer, code_generator::CodeGeneratorConfig,
    typescript::Installer, bincode::BincodeSerializer,
};

fn main() -> anyhow::Result<()> {
    let mut tracer = Tracer::new(Default::default());
    tracer.trace_type::<cross_platform_crux::Event>()?;
    let registry = tracer.registry()?;

    let out_dir = "bindings/ts";
    std::fs::create_dir_all(out_dir)?;

    Installer::new(out_dir)
        .with_serializer(BincodeSerializer::default())
        .install(&registry)?;

    Ok(())
}
```

# wasm-pack

- Generiert WASM-Modul und JS-Glue-Code

```
static CORE: LazyLock<Bridge<EmailApp>>
= LazyLock::new(|| Bridge::new(Core::new()));

#[cfg_attr(target_family = "wasm", wasm_bindgen::prelude::wasm_bindgen)]
#[must_use]
pub fn process_event(data: &[u8]) -> Vec<u8> {
    match CORE.process_event(data) {
        Ok(effects) => effects,
        Err(e) => panic!("{}"),  

    }
}
```

cross\_platform\_crux.d.ts

```
export function process_event(data: Uint8Array): Uint8Array;
```

# Flutter Rust Bridge

- Rust-Code in Flutter-Projekt integrieren

frb\_cross\_platform\_core.rs

```
static CORE_CELL: OnceLock<EmailAppCore> = OnceLock::new();

#[frb(sync)]
pub fn init_app_and_get_view_model() -> ViewModel {
    let core = cross_platform_rust::create_core();
    let view_model = core.view();
    let _ = CORE_CELL.set(core);
    view_model
}

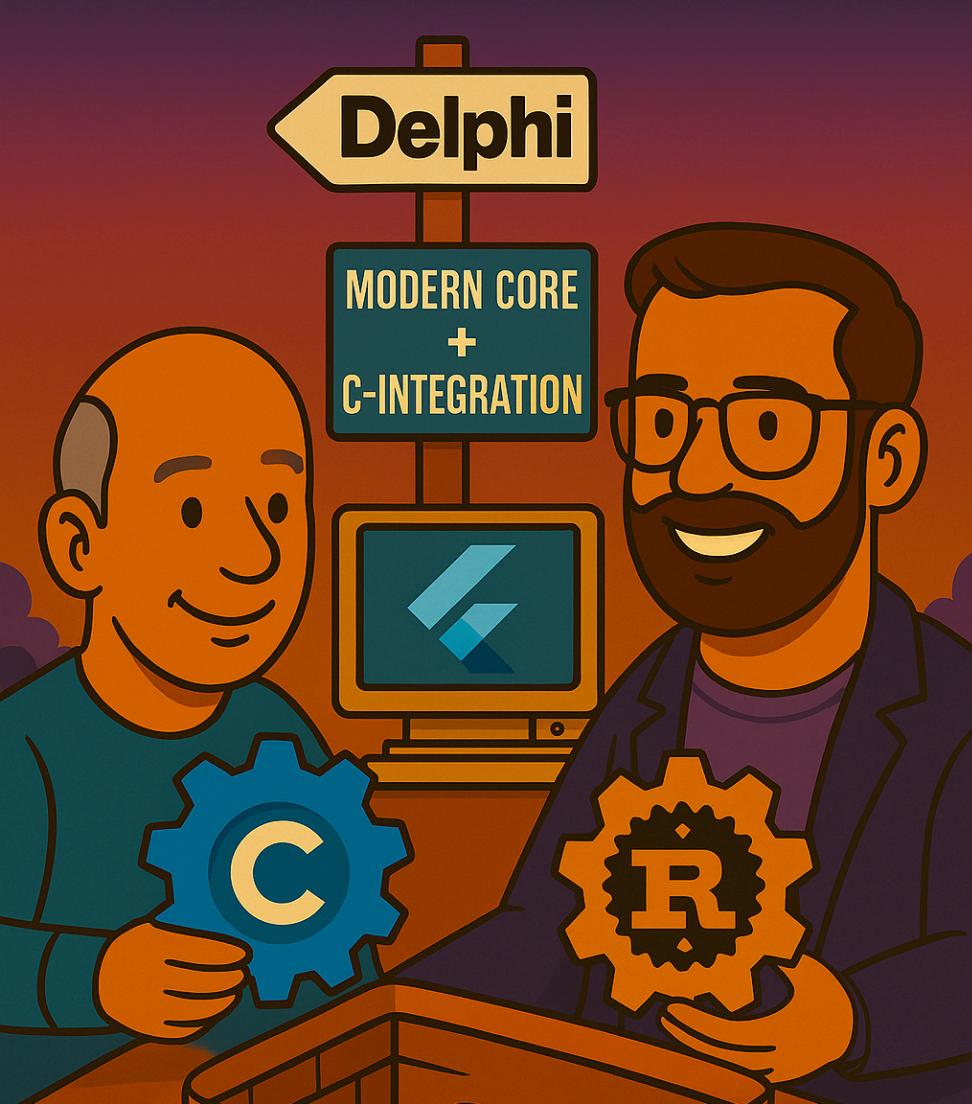
#[frb(mirror(Event))]
#[frb(dart_metadata="freezed")] // generates dart classes
pub enum _Event {
    ChangeEmail(String),
    EmailSent(bool),
}
```

```
import 'package:cross_platform_rust_app/src/rust/api/frb_cross_platform';
import 'package:cross_platform_rust_app/src/rust/frb_generated.dart';

Future<ViewModel> initRustCore() async {
    await RustLib.init();

    final viewModel = initAppAndGetViewModel();

    return viewModel;
}
```



# Es kam einmal ein Kunde

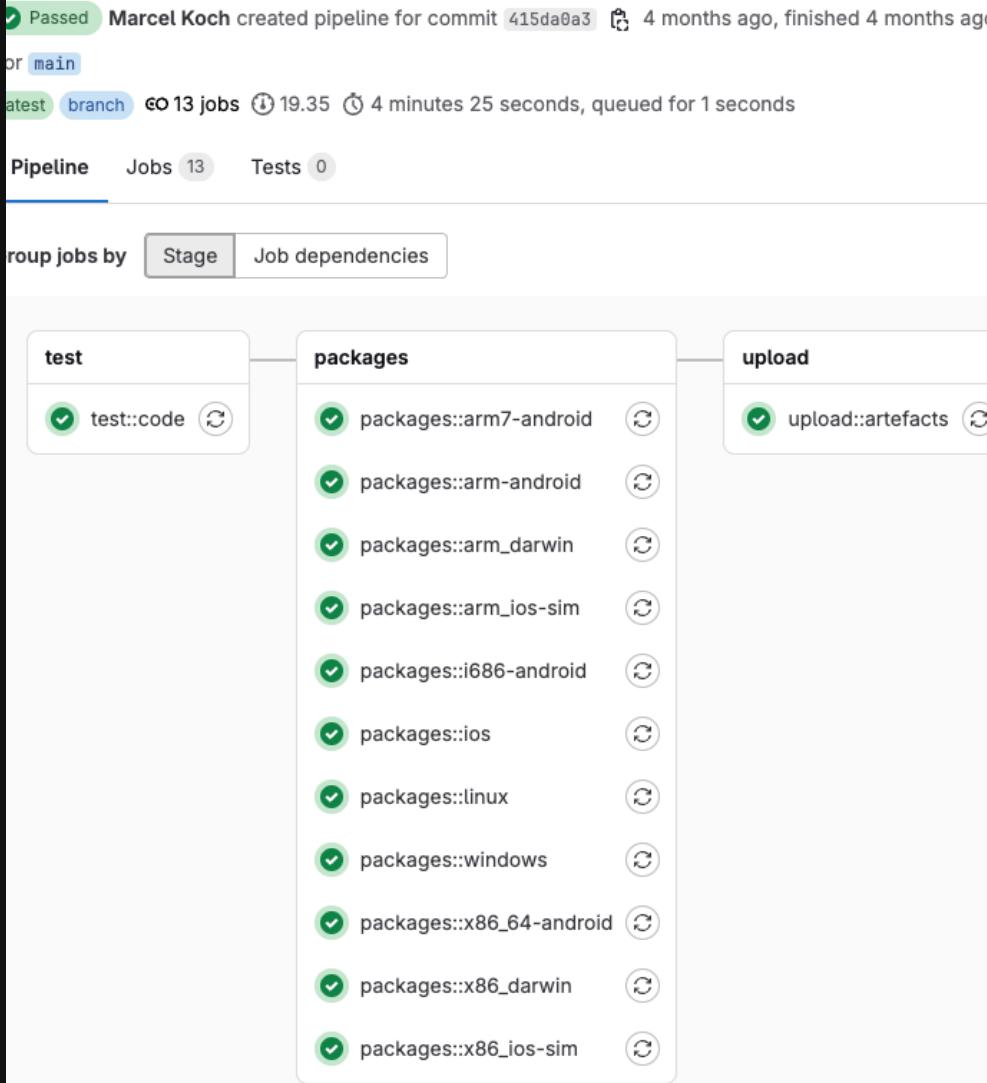
- C
  - War gesetzt
  - Ist gesetzt
- Worauf bauen wir die UI?
  - Flutter
  - React Native

# Erfahrungen aus der Praxis

- C
- uniffi - c# ist nicht so einfach
- Mehr als crux liefert
- CI/CD

# Um C zu unterstützen...

- bauen wir libs vor
- unterstützen wir 11 targets
- binden wir die libs in die `build.rs` mit eigenem Tool `grab-n-spread` ein
- C-Integration an sich kein Problem
- Cross compiling sehr wohl
  - Windows: .lib falsche Berechnung
  - dockcross
  - osxcross



# Mehr als crux mitbringt

- I18N (integriert im Kompilat)
- Labels
- Logging

# Zusammenfassung

- Architektur trennt langlebigen Core von kurzlebiger UI
- Core übernimmt soviel wie möglich: Zustand, Logik, Validierung
- Mögliche Lösung: Rust, crux, Flutter, FRB



# Danke!



[www.marcelkoch.net](http://www.marcelkoch.net)

