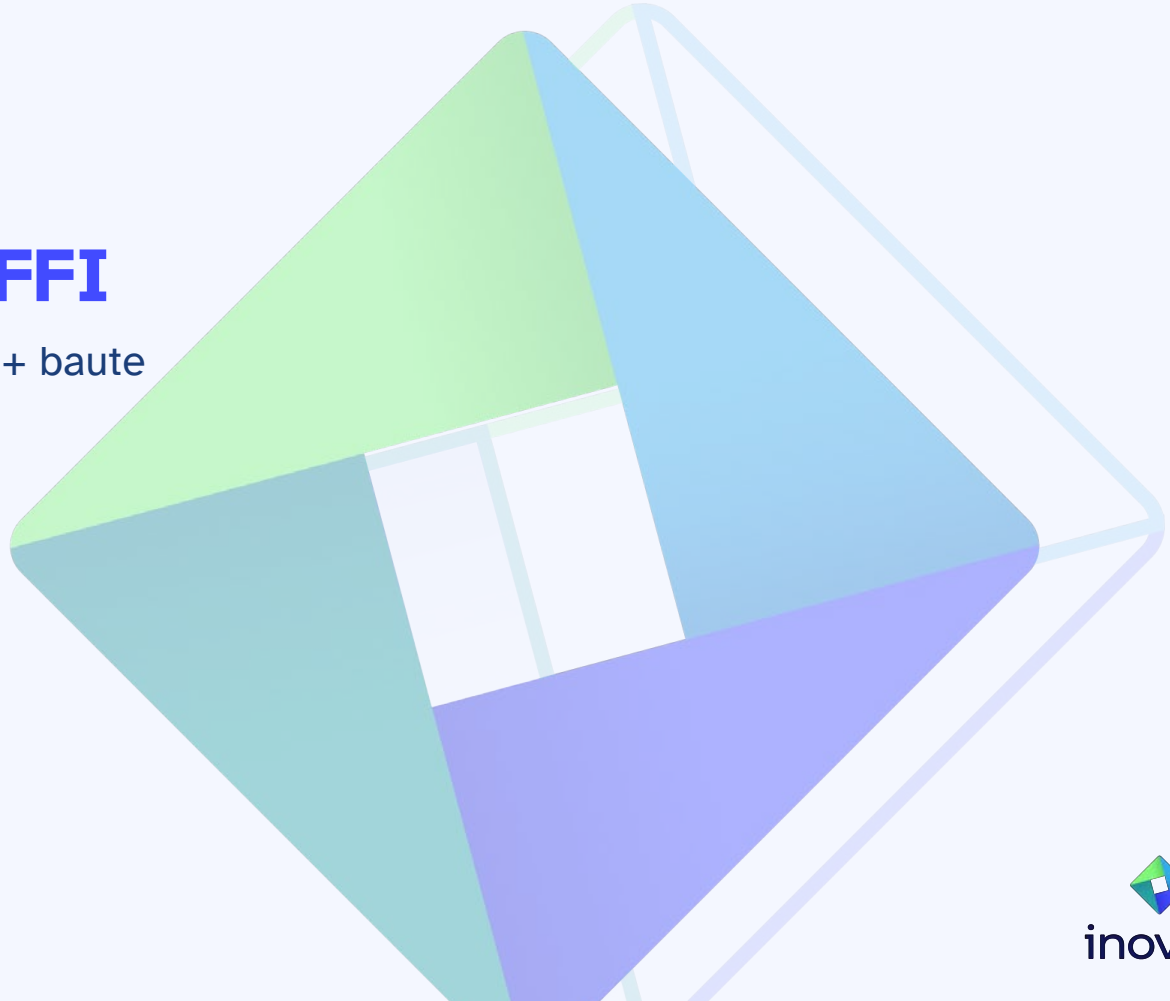
# Async Rust und FFI

oder wie ich eine Runtime in C++ baute



**Florian Limberger**
Rust Meetup Köln

inovex

# Florian Limberger

Embedded Software Engineer

- Embedded Linux
- Echtzeit-Systeme

Florian Limberger

flimberger



inovex

# Eine rostige Agenda

- WTF?!
- Wie funktioniert eigentlich Async Rust?
- Async FFI
- Eine Async Runtime in C++

inovex


# Warum sollte man eine Rust Async Runtime in C++ schreiben wollen?!



inovex

# Warum?

Firmen-interne Politik:

- Neue Komponente soll in Rust geschrieben werden
- Vorhandener Code muss integriert werden
  - TypeScript (+ Cordova)
  - C++
  - C#
  - ...
- Code soll async sein
  - Verbesserte Leistung bei I/O-intensiven Aufgaben
  - Abbildung von Nebenläufigkeit

Realistischer Anwendungsfall: Rust Plugins in C++-Anwendungen

# Wie funktioniert eigentlich Async Rust?

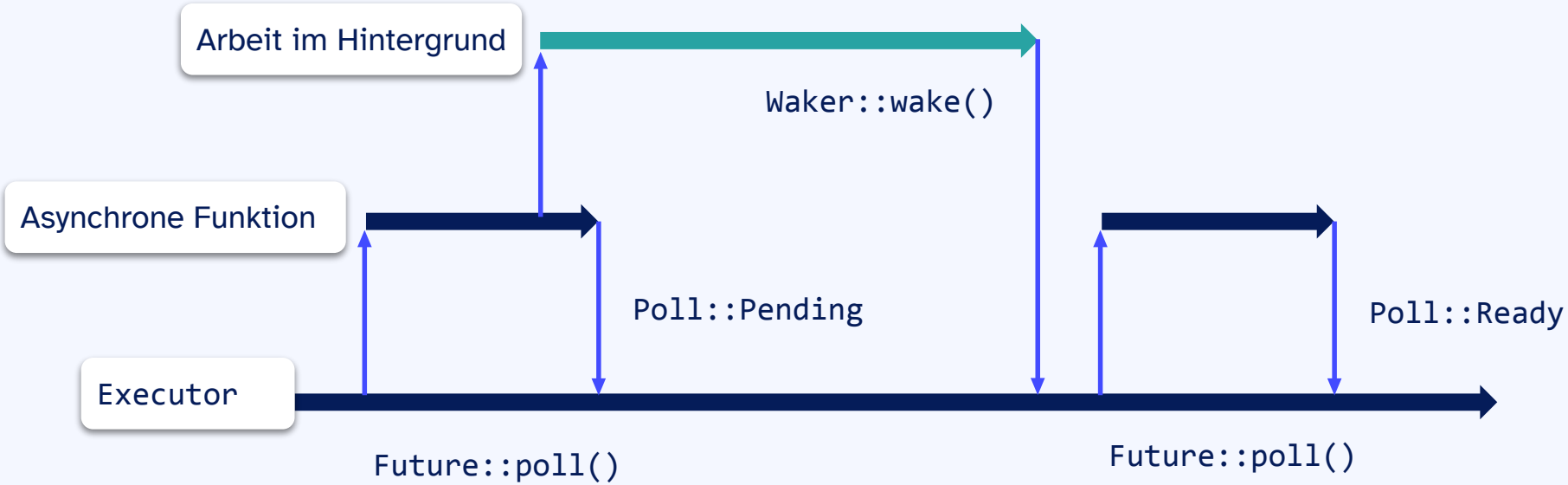


inovex

# Async Rust unter der Haube

- `async`/`.await`: syntactic sugar für asynchrone Primitiven
- Runtime (nicht aus der Standard-Bibliothek)
  - Tasks
- Primitiven aus der Standard-Bibliothek:
  - `trait core::future::Future`
  - `enum core::future::Poll<T>`
  - `struct core::task::Context`
  - `struct core::task::Waker`

# Zusammenarbeit der Komponenten






inovex

## Syntactic Sugar: `async` und `.await`

- `async` Funktionen geben Futures zurück
- `.await`-Statements werden in State Machines umgewandelt, welche `Future::poll()` aufrufen
  - Compiler transformiert Code
  - hier findet die eigentliche “Magie” statt

# Transformation von `async`

```rust
async fn open_async(p: AsRef<Path>) -> Result<File, Error> {
    // implementation details
    Ok(file)
}

async fn use_file() -> Result<(), Error> {
    let file = open_async("foo.bar").await?;
    // use the file here
    Ok(())
}
```

# Transformation von async

```rust
async fn open_async(path: AsRef<Path>)
    -> Result<File, Error>
{
    // implementation details
    Ok(file)
}

async fn use_file()
    -> Result<(), Error>
{
    let file = open_async("foo.bar").await?;
    // use the file here
    Ok(())
}
```

```rust
fn open_async(path: AsRef<Path>)
    -> Future<Result<File, Error>>
{
    // implementation details
    Ok(future)
}

fn use_file()
    -> Future<Result<(), Error>>
{
    // more in a minute
}
```



inovex

# Das Future Trait

```rust
enum Poll<T> {
    Ready(T),
    Pending,
}

pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, context: &mut Context<'_>) -> Poll<Self::Output>
}
```



inovex

# Transformation von `.await`

```
01 async fn use_file()
02     -> Result<(), Error>
03 {
04     let file =
05         open_async(“foo.bar”)
06         .await?;
07     // use the file here
08     Ok(())
09 }
```

```
01 struct MyFuture(Future<Result<File, Error>>)
02
03 fn use_file(ctx: &mut Context) -> Future<Result<(), Error>> {
04      MyFuture(open_async(“foo.bar”))
05 }
06
07 impl Future for MyFuture {
08     type Output = Result<(), Error>;
09
10     fn poll(self: Pin<&mut Self>, ctx: &mut Context)
11         -> Poll<Self::Output>
12     {
13         let poll_result = future.poll(ctx);
14         match poll_result {
15             Poll::Ready(file_result) => {
16                 // use the file here
17                 Poll::Ready(Ok(()))
18             },
19             Poll::Pending => Poll::Pending,
20         }
21     }
22 }
```

Eventuelle Fehler werden hier behandelt



inovex

# Context und Waker

Context stellt lediglich einen Waker bereit:

```
01 struct Context<'a> { /* private */ }
02
03 impl<'a> for Context<'a> {
04     fn waker(&self) -> &'a mut Waker { /* … */ }
05 }
```

Ein Waker dient zum Aufwecken der Runtime:

```
01 struct Waker { /* private */ }
02
03 impl for Waker {
04     fn wake(self) { /* … */ }
05     fn wake_by_ref(&self) { /* … */ }
06 }
```

# Beispielhafte Anwendung von Waker

```
01 fn request_file_open(path: Path, result: &mut Option<File, Error>>, on_done: impl FnOnce() + 'static);
02
03 struct FileFuture { path: Path, result: Option<Result<File, Error>>, }
04
05 impl Future for FileFuture {
06     fn poll(self: Pin<&mut Self>, ctx: &mut Context) {
07         match self.result {
08             None => {
09                 let waker = ctx.waker().clone();
10                 request_file_open(self.path, &mut self.result, Box::new(move || { waker.wake(); }));
11                 Poll::Pending
12             },
13             Some(result) => Poll::Ready(result),
14         }
15     }
16 }
17
18 fn open_async(ctx: &mut Context, path: AsRef<Path>) -> Future<Result<File, Error>> {
19     FileFuture { path, Option::None }
20 }
```

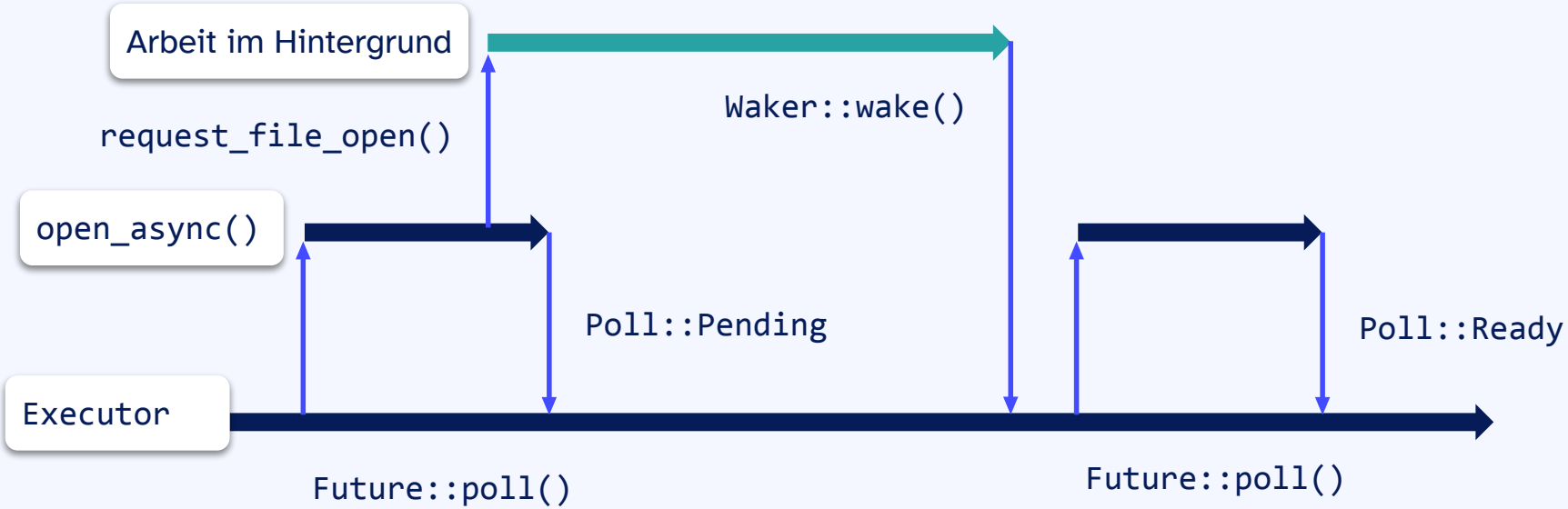
PSEUDOCODE

Waker wird im Callback aufgerufen



inovex

# Zusammenarbeit der Komponenten






inovex

# Executor

- Teil der Runtime
  - *Nicht* Teil der Standardbibliothek, sondern extern
- async Funktionen geben Futures zurück
- Irgendjemand muss `Future::poll()` am Eintrittspunkt (a.k.a. `main`) aufrufen
- Dies übernimmt der **Executor**

# Async Runtimes

Nicht Teil der stdlib, sondern von der Community beigesteuert:

- tokio
- async-std
- smol

# Async FFI



inovex

# `async-ffi` Crate

- Futures sind standardmäßig nicht FFI-kompatibel
- `async-ffi` crate: https://crates.io/crates/async-ffi
  - Stellt FFI-kompatible Wrapper für Futures bereit
  - *Primär* für Plugins gedacht, also Rust□Rust FFI
- Overhead
  - bei jedem FFI-Übergang werden Objekte alloziert
  - Wrapper-Objekte liegen teilweise auf dem Stack und *müssen* kopiert werden

# FFI-Repräsentation einer Future

```rust
#[repr(C)]
pub struct FfiFuture<T> {
    fut_ptr: *mut (),
    poll_fn: unsafe extern "C" fn(
            fut_ptr: *mut (),
            context_ptr: *mut FfiContext
        ) -> FfiPoll<T>,
    drop_fn: unsafe extern "C" fn(*mut ()),
}

impl<T> Future for FfiFuture<T> {
    type Output = T;

    fn poll(self: Pin<&mut Self>, ctx: &mut Context<'_>) -> Poll<Self::Output) {
        unsafe {
             (self.poll_fn)(self.fut_ptr, FfiContext::from(ctx))
        }
    }
}
```

Jede Menge Zeiger und unsafe!



inovex

# FFI-Repräsentation von Context und Waker

```rust
#[repr(C)]
pub struct FfiContext {
    waker: *const FfiWaker,
}

#[repr(C)]
pub struct FfiWaker {
    vtable: *const FfiWakerVTable,
}

#[repr(C)]
pub struct FfiWakerVTable {
    clone: unsafe extern "C" fn(*const FfiWaker) -> *const FfiWaker,
    wake: unsafe extern "C" fn(*const FfiWaker),
    wake_by_ref: unsafe extern "C" fn(*const FfiWaker),
    drop: unsafe extern "C" fn(*const FfiWaker),
}
```

impl Clone for FfiWaker

impl Drop for FfiWaker



inovex

# Anwendungsbeispiel

```
01 #[repr(C)]
02 struct FfiCallback {
03     async_callback: unsafe extern "C" fn(i32) -> FfiFuture<Foo>,
04 }
05
06 #[no_mangle]
07 pub unsafe extern "C" mylib_run_async(callback: *const FfiCallback) -> FfiFuture<()>
{
08     async {
09         let n = run_async().await
10         ((*callback).async_callback)(n).await;
11     }
12     .into_ffi()
13 }
```

Just works!



inovex

# Die Runtime in C++



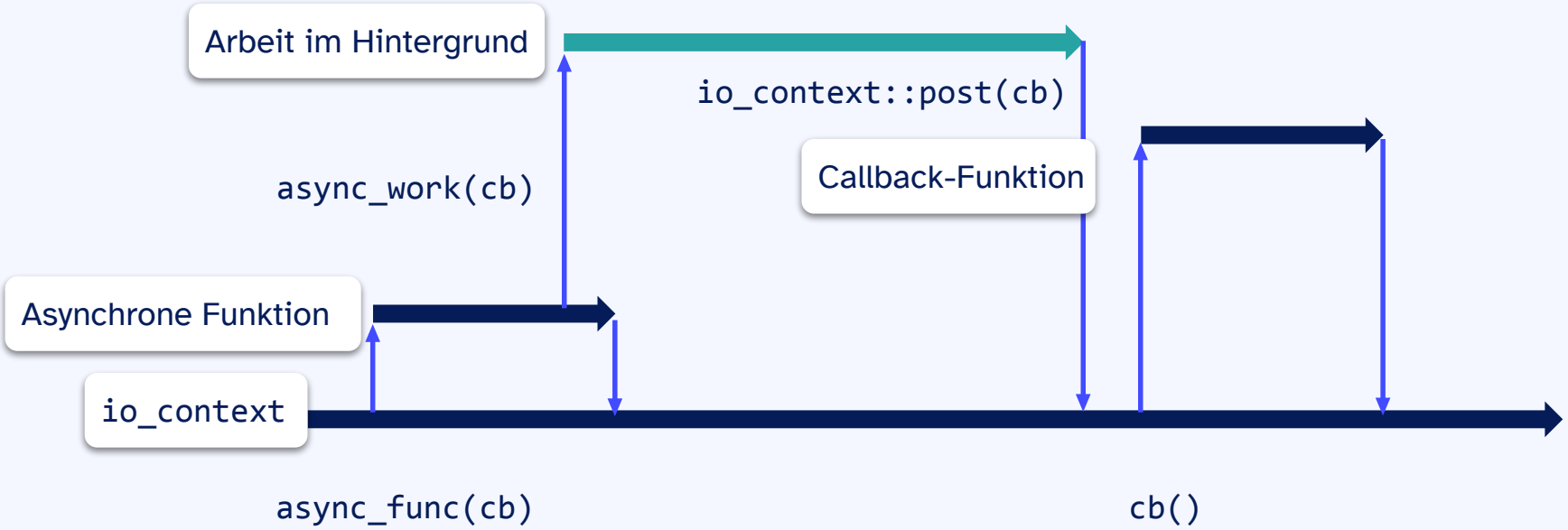
inovex

# Randbedingungen

- Boost.Asio
  - Häufig verwendet (im Corporate-Umfeld tarnen sich so viele Bibliotheken als eine einzige)
  - Ermöglicht Verwendung von Boost.Beast als HTTP Client
- Callback-basiertes Async

**Unterschiedliche Async-Prinzipien müssen aneinander angepasst werden!**

# Asynchronizität in Boost.Asio






inovex

# Komponenten

- Waker: Implementierung für den Rust Waker
- Task: Ausführeinheit im Executor
- Executor: queued Tasks, ruft `Future::poll()` auf
  - Eigentliche Asynchronität wird an Boost.Asio ausgelagert
- Glue-Code
  - Wrapper für Futures C++ → Rust
  - Wrapper für Futures Rust → C++
  - Kopplung von Boost.Asio Async Code und Rust Async Code

# Implementierung: Übersicht

- Boost.Asio Asynchronität nutzt Callback-Funktionen
- `Task` speichert eine `RustFuture` und eine Callback-Funktion
  - Task leitet `poll()` Funktion an `RustFuture` weiter
  - Wenn Ready wird das Resultat an die Callback-Funktion übergeben
- `Executor` erstellt einen Task, ruft `Task::poll()` auf
  - Wenn Pending zurückgegeben wird, wird der Task in einer Queue gespeichert
- Wenn Rust `Waker::wake()` aufruft, wird der Task an Boost.Asio zur asynchronen Ausführung übergeben

# Implementierung: Beispiel C++ → Rust

```
01 asio::io_context io_context{};
02 asyncrt::Executor executor{io_context};
03
04 auto data_access = std::make_unique<DataAccess>(io_context);
05
06 auto* lib = ::mylib_alloc(&data_access);
07
08 io_context.dispatch([&executor, &lib]() {
09     auto future = ::mylib_do_foo(42);
10     executor.await(std::move(future), [](bool const& result) {
11         std::cout << "received " << result << " from mylib" << std::endl;
12     });
13 });
14
15 io_context.run();
16
17 ::mylib_free(lib);
```

asynchrone Rust Funktion



inovex

# Implementierung: Beispiel Rust → C++

```
::FfiFuture<Container*> get_data() {
    auto promise = asyncrt::Promise<std::string>{};
    auto future = promise.get_future();
    http::get(io_context, "api.example.com", "/v1/foo",
              [promise = std::move(promise)](std::string const& result) mutable {
                  promise.set_value(result);
              });
    return asyncrt::make_cpp_future<Container*>([future = std::move(future)](
                                                    ::FfiContext* context) mutable {
        if (future.is_ready()) {
            auto* p = new Container{future.value()};  // implements Drop
            return asyncrt::make_poll_status(static_cast<Container*>(p));
        }
        auto waker = std::shared_ptr{context->waker->clone()};
        future.await([waker]() {
            waker->wake_by_ref(waker);
        });
        return asyncrt::make_poll_status<Container*>(asyncrt::PollStatus::Pending);
    });
}
```

Async Bridge Glue Code



inovex

# Herausforderungen

- Quasi keine Typsicherheit
  - FFI-Code ist (fast) reines C
  - keine automatisches Ownership-Tracking, keine Smartpointer
  - FFI-Schnittstelle muss manuell aktuell gehalten werden
- Manuelle Speicherverwaltung
  - Jede Menge memory-leaks
- Keine Compiler-Unterstützung
  - Futures und Poll-State Machine müssen manuell geschrieben werden
- Interessantes ownership-Konzept in Rust: `Waker::wake()` konsumiert sich selbst

# Fazit

- Es funktioniert tatsächlich!
- Die `poll()`-getriebene Ausführung ist sehr flexibel
- Während der Entwicklung gibt es *jede Menge* Speicherfehler
  - `valgrind` und `-fsanitize=address` sind die besten Freunde während der Entwicklung
  - langes, langes Debuggen (`printf()` ftw!)

https://github.com/inovex/async-rust-ffi-runtimes



inovex

# Vielen Dank!

**Florian Limberger**
**Embedded Software Engineer**

florian.limberger@inovex.de

Ludwig-Erhard-Allee 6
76131 Karlsruhe

Florian Limberger

flimberger



inovex

# Schön, dass du hier bist – bleib in Kontakt!

We’re hiring!

Podcast


Überall, wo es Podcasts gibt: Spotify, Overcast, Pocket Casts, ...

Soziale Medien

Twitter: @inovexgmbh

Insta: @inovexlife

LinkedIn: @inovex GmbH



inovex

# Implementierung: Task

```
template <typename T, typename F>
class Task : public detail::TaskBase {
public:
    Task(RustFuture<T> future, F&& callback, Executor& executor, uint64_t id)
        : TaskBase{executor, id},
          m_future{std::move(future)},
          m_callback{std::move(callback)} {}
protected:
    [[nodiscard]] PollStatus poll_impl(Executor& executor) override {
        auto poll = m_future.poll(this->get_context());
        if (poll.status == PollStatus::Ready) {
            m_callback(poll.value);
        }
        return poll.status;
    }
private:
    RustFuture<T> m_future;
    F m_callback;
};
```



inovex

# Implementierung: Executor

```
class Executor {
public:
    Executor(boost::asio::io_context& ioCtx);

    template <typename T, typename F>
    void await(RustFuture<T> future, F&& callback) {
        auto task = std::make_shared<detail::Task<T, F>>(
            std::move(future), std::forward<F>(callback), *this, m_last_task_id++);
        auto done = task->poll(*this);
        if (!done) {
            m_tasks.emplace_back(std::move(task));
        }
    }

    // used by the task when it was woken
    void ready(detail::TaskBase& task);

private:
    std::vector<std::shared_ptr<detail::TaskBase>> m_tasks{};
    boost::asio::io_context& m_ioctx;
    uint64_t m_last_task_id = 0;
};
```



inovex

# Implementierung: Waker

```
class Waker : public ::FfiWakerBase {
public:
    Waker(Executor& executor, TaskBase& task);

private:
    ::FfiWakerBase const* clone_impl() const;
    void wake_impl() const;
    void wake_by_ref_impl() const;
    void drop_impl() const;

    Executor& m_Executor;
    TaskBase& m_task;
};

void Waker::wake_by_ref_impl() const {
    m_Executor.ready(m_task);
}
```