Strongly Typed Financial Software

- Michael Snoyman
- VP Software, FP Block
- Rustikon 2025, Poland

RUSTIKON
Powered by SoftwareMill

Code for this presentation available at:

https://github.com/snoyberg/rustikon-2025

We're hiring! Catch me at the afterparty for more info.

FP Block

What is Rust?

Systems programming language

High performance

No garbage collector

Zero cost abstractions

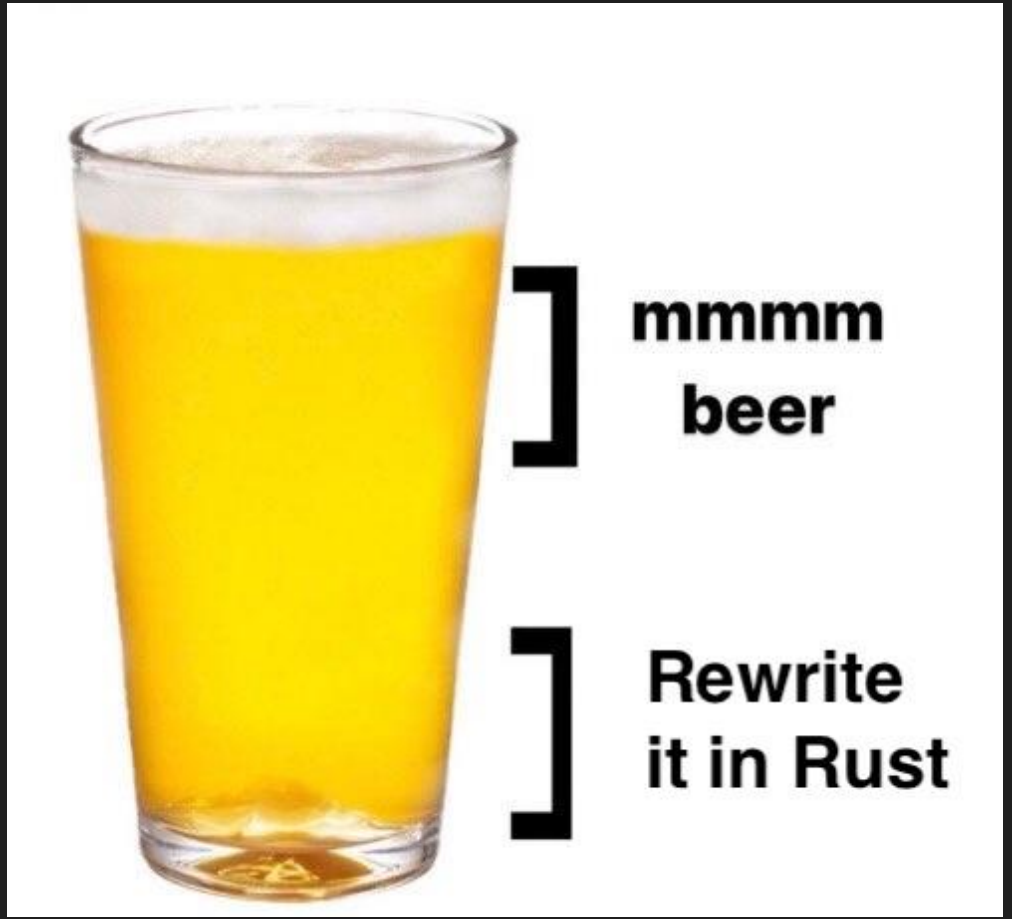
Procedural

But what do I like in Rust?

- Statically typed
- Strongly typed (more on this shortly)
- Immutable-by-default
- Pattern matching
- Built in test suites
- Property testing
- And much more! Everyone can have their own favorite parts of Rust

FP Block

Remember to Rust in moderation



FP Block

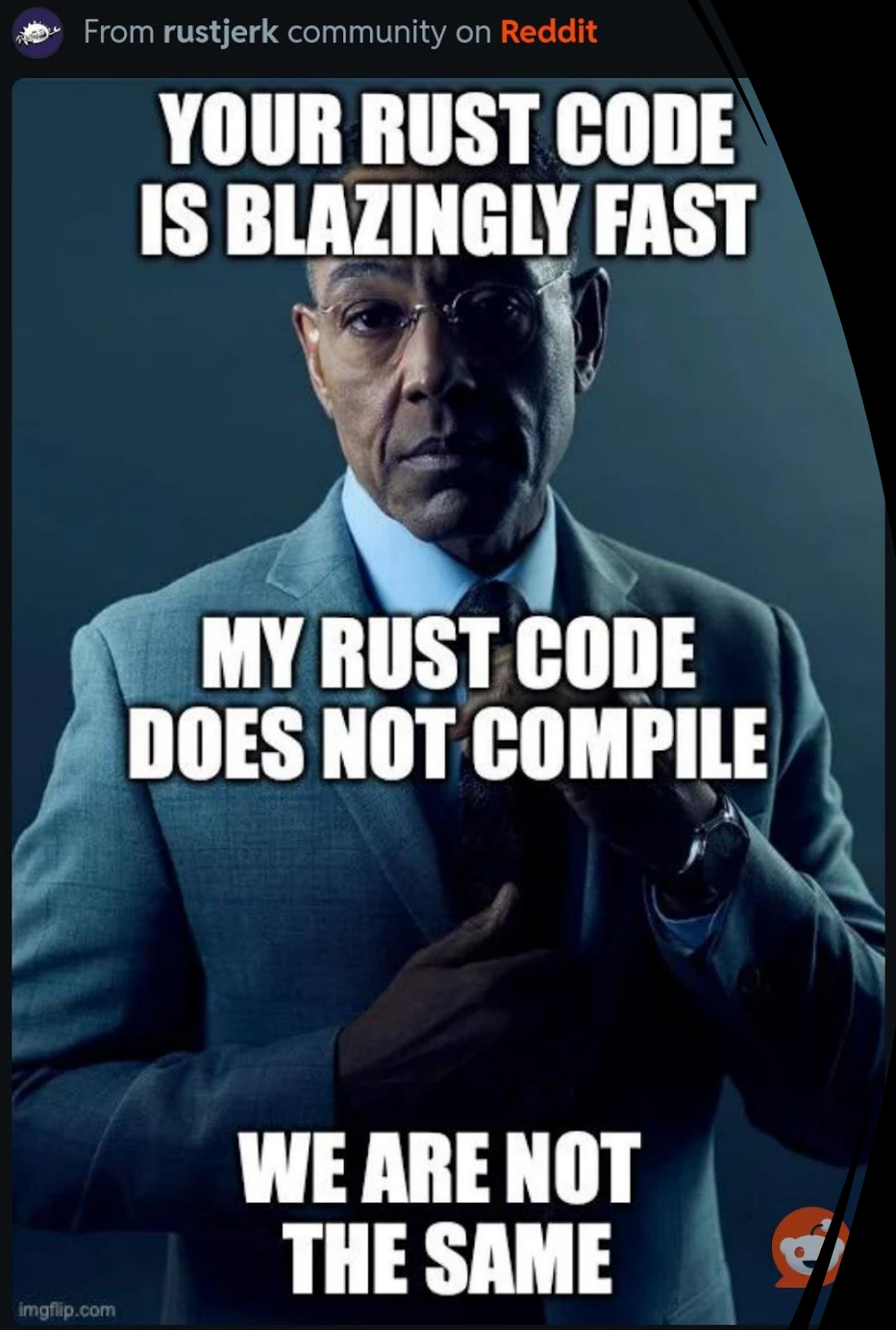
Static vs strong typing

Static typing: types are checked at compile time.	• Examples: C, C++, Java, Rust
Dynamic typing: types are checked at runtime.	• Examples: Python, JavaScript, PHP
Strong typing: the language makes it easy to express complicated invariants in the type system.	• Examples: Rust, Haskell, OCaml
Weak typing: the language does not make complicated invariants easy to express.	• Examples: C, Java

What are “strong typing” features?

Sum types	Newtypes	Structural polymorphism (generics)	Checked parametric polymorphism (traits)
Encapsulation (smart constructors)	Nominal typing (versus, e.g. TypeScript’s structural typing)	Associated types	Parameterized traits

FP Block



Compiling should fail!

- Strong types can prevent classes of bugs.
- We want to code in a way that leverages these strong types.
- Result will be failed compilation!
- This is a good thing! It tells us exactly where to fix our code.

Financial software concerns

- Precision matters more than many other domains
  - Usually involves specific laws around rounding, for example
- Mistakes can lead to loss of money and/or jail time
- Easy to mix up many different “numbers”
  - E.g. don’t accidentally add a price in dollars and another in euros
- Some values can never be 0, others can never be negative, others can be both

FP Block

What's wrong with this code?

```
fn main() {
    let price_apple = 1.3;
    let price_banana = 0.8;
    let apples = 5.0;
    let bananas = 9.0;
    let total = (price_apple + price_banana) * (apples + bananas);
    println!("Your total comes out to {total}");
}
```

- Floating point representation of an integral value
- Floating point rounding may lead to incorrect results
- The math is wrong! We shouldn't add prices together like that

Can we make these kinds of mistakes impossible?

FP Block

Add some newtypes

- Use the Decimal datatype
- Newtype wrappers to represent price versus the total in USD
- Even better: put the newtypes in their own module, force only safe construction (smart constructors)
- Do not define incorrect operations e.g.
  - Cannot multiply prices
  - Cannot add prices
- The code no longer compiles, that's great!

```
use rust_decimal::Decimal;
                                                                cannot add `Price` to `Price`
/// Price of an item given in USD                                                (E0369)
struct Price(Decimal);

/// An amount in US Dollars
struct Usd(Decimal);

impl Price {
    fn calc_total(&self, quantity: u32) -> Usd {
        Usd(self.0 * Decimal::from(quantity))
    }
}

fn main() {
    let price_apple = Price(1.3);
    let price_banana = Price(0.8);
    let apples = 5;
    let bananas = 9;
    let total = (price_apple + price_banana) * (apples + bananas);
    println!("Your total comes out to {total}");
}
```

FP Block

And make it compile

```rust
impl std::ops::Add for Usd {
    type Output = Usd;

    fn add(self, rhs: Self) -> Self::Output {
        Usd(self.0 + rhs.0)
    }
}

impl std::fmt::Display for Usd {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "${:.02}", self.0)
    }
}

fn main() {
    let price_apple = Price("1.3".parse().unwrap());
    let price_banana = Price("0.8".parse().unwrap());
    let apples = 5;
    let bananas = 9;
    let total = price_apple.calc_total(apples) + price_banana.calc_total(bananas);
    println!("Your total comes out to {total}");
}
```

- Add in operations (Add and Display) where they make sense
- Use the correct data types in main, as prompted by the compiler
- The Usd data type knows how to display correctly
- Cannot accidentally add the prices together

FP Block

Taking it too far

```rust
struct Apple;
struct Banana;

struct Price<Item> {
    price: Decimal,
    _phantom: PhantomData<Item>,
}

impl<Item> Price<Item> {
    fn from_static(_: Item, price: &'static str) -> Self {
        Price {
            price: price.parse().unwrap(),
            _phantom: PhantomData,
        }
    }

    fn calc_total(&self, quantity: Quantity<Item>) -> Usd {
        Usd(self.price * Decimal::from(quantity.amount))
    }
}

struct Quantity<Item> {
    amount: u32,
    _phantom: PhantomData<Item>,
}

impl<Item> Quantity<Item> {
    fn new(_: Item, amount: u32) -> Self {
        Quantity {
            amount,
            _phantom: PhantomData,
        }
    }
```

- There’s always room to keep making stronger types
- At some point, there are diminishing returns
- Don’t add in extra type safety for fun
- Add it where:
  - You’re preventing a likely bug from occurring
  - The extra effort to get this type safety is warranted by the protection
- Concretely: I'd *not* write this kind of code, even though it’s more type safe

FP Block

The rest of this talk

Everyone loves code, right?

Let’s go through some!

We’re going to build a “spot swap” application

Server

- Tracks a user’s balance of USD vs Euros
- Allows swaps between them
- Admin can give away free money (yay!)

Client (in Rust, of course)

- View balance
- Trade dollars for euros (or vice versa)

FP Block

Caveat emptor!

- Translation: buyer beware
- The code we'll be looking at is *not* the best approach possible
- I've taken some approaches to show off Rust type abilities
- Feel free to ask on any point whether I'd recommend it in production
- Also: I legitimately made a bunch of errors while writing this code that the type system and test suite caught

FP Block

Numerics

Need to represent money

Want to use a decimal type

In real code: please use an existing library!

We'll write our own

https://github.com/snoyberg/rustikon-2025/tree/main/packages/numeric

FP Block

Encapsulation

- Define our UnsignedDecimal
- Simple wrapper around u128
- Enforces invariants around decimal handling
- Type is defined in a private submodule
- We expose raw operations to the rest of the crate
- Then we provide a nicer API within the crate for public consumption
- But there aren't really any invariants to enforce here...

```rust
mod private {
    /// Stored with 6 digits of precision
    #[derive(PartialEq, Eq, PartialOrd, Ord, Clone, Copy)]
    pub struct UnsignedDecimal {
        value: u128,
    }

    impl UnsignedDecimal {
        pub(crate) fn from_raw_value(value: u128) -> Self {
            UnsignedDecimal { value }
        }
        pub(crate) fn get_raw_value(&self) -> u128 {
            self.value
        }
    }
}
```

FP Block

Signed decimals

- Same private submodule approach
- Builds on UnsignedDecimal, adds a negative field
- Problem: now there are two representations of 0!
- Solution: enforce an invariant
- from_raw_value enforces the invariant
- No other part of the codebase can create a SignedDecimal

```rust
mod private {
    use crate::UnsignedDecimal;

    /// A signed version of [UnsignedDecimal]
    #[derive(PartialEq, Eq, PartialOrd, Ord, Clone, Copy)]
    pub struct SignedDecimal {
        value: UnsignedDecimal,
        // Invariant: negative must be false whenever value is 0
        negative: bool,
    }

    impl SignedDecimal {
        pub(crate) fn from_raw_value(value: UnsignedDecimal, negative: bool) -> Self {
            SignedDecimal {
                value,
                negative: negative && value.get_raw_value() != 0,
            }
        }
        pub(crate) fn get_raw_value(&self) -> UnsignedDecimal {
            self.value
        }
        pub(crate) fn is_negative(&self) -> bool {
            self.negative
        }
    }
}
```

FP Block

Positive decimals

- Lots of financial operations want to ensure "greater than 0"
- Example: price of assets must always be non-zero
- New wrapper around UnsignedDecimal
- New invariant to implement: reject 0
- Return a Result from new representing the possibility of a 0
- new is a **smart constructor**

```rust
mod private {
    use anyhow::Result;

    use crate::UnsignedDecimal;

    /// A version of [UnsignedDecimal] which disallows the value 0.
    #[derive(PartialEq, Eq, PartialOrd, Ord, Clone, Copy)]
    pub struct PositiveDecimal {
        // Invariant: can never be 0
        value: UnsignedDecimal,
    }

    impl PositiveDecimal {
        /// Generate a new value, checking that the input is not 0.
        pub fn new(value: UnsignedDecimal) -> Result<Self> {
            anyhow::ensure!(value.get_raw_value() != 0, "PositiveDe
            Ok(PositiveDecimal { value })
        }

        /// Get the raw unsigned value.
        pub fn get_unsigned(&self) -> UnsignedDecimal {
            self.value
        }
    }
}
```

FP Block

Parsing with smart constructors

- No explicit data validation
- Leverages existing parse logic for UnsignedDecimal
- Invariant is automatically enforced via PositiveDecimal::new
- By hiding internals of PositiveDecimal, we know the only way to construct it is via the smart constructor
- One example, the rest of the numerics crate shows others

```rust
impl FromStr for PositiveDecimal {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        s.parse().and_then(PositiveDecimal::new)
    }
}
```

FP Block

Assets

- Our code will need to distinguish between USD and EURO values
- We *could* use our Decimal types directly for this
- Downsides
  - Very easy to use USD for EUROs or vice-versa.
    - I made this mistake many times while writing the code, the compiler saved me.
  - No tagging in the on-the-wire representation to disambiguate
    - In Yesod, I call this the boundary issue.
- Instead, we'll have tagged datatypes to represent assets

https://github.com/snoyberg/rustikon-2025/tree/main/packages/common

FP Block

Parameterized Types, Phantoms, and tagging

```rust
#[derive(Debug, PartialEq, Eq, PartialOrd, Ord, Default, Clone)]
pub struct UnsignedAsset<T> {
    value: UnsignedDecimal,
    _phantom: PhantomData<T>,
}

#[derive(Debug, PartialEq, Eq, PartialOrd, Ord)]
pub struct PositiveAsset<I> {
    value: PositiveDecimal,
    _phantom: PhantomData<I>,
}
```

- We want to distinguish different kinds of assets
- Use a type parameter to create different types
- Also carry over Unsigned vs Positive (we could also do Signed)
- Need to use PhantomData – no runtime representation
- Automatically get validation guarantees of underlying data type
- Compiler can now distinguish between dollars and euros

FP Block

Asset Trait and Macro

- Trait (ad-hoc polymorphism) for any assets
- Helper macro to generate concrete datatypes
- Requires some upfront setup
- After that, adding new assets is trivial

```rust
/// Any type that represents an asset type.
pub trait Asset: Ord + std::fmt::Debug + Default {
    fn as_str() -> &'static str;
}

macro_rules! make_asset {
    ($i:ident, $name:expr) => {
        #[derive(PartialEq, Eq, PartialOrd, Ord, Debug, D
        pub struct $i;
        impl Asset for $i {
            fn as_str() -> &'static str {
                $name
            }
        }
    };
}

make_asset!(Usd, "USD");
make_asset!(Euro, "EURO");
// Not needed, just for fun
make_asset!(Bitcoin, "BTC");
```

FP Block

Some type safety!

- This code doesn't compile
- Not compiling is a GOOD THING!

```
}                          cannot add `asset::UnsignedAsset<asset::Euro>` to
                                       `asset::UnsignedAsset<asset::Usd>`
#[test]                                                                (E0369)
fn cannot_add_dollars_euros() {
    let dollars: UnsignedAsset<Usd> = "1000USD".parse().unwrap();
    let euros: UnsignedAsset<Euro> = "1000EURO".parse().unwrap();
    dollars + euros
}
```

FP Block

Representing prices

- Want to discuss the price of the **base asset** (e.g., apples) in terms of the **quote asset** (e.g., dollars)
- Easy to make mistakes about this when dealing with Forex (is USD or EURO the base asset?)
- Price<Usd, Euro> means "how many EUROs to buy 1 USD?"
- Price<Euro, Usd> means "how many USDs to buy 1 EURO?"
- No need to check for divide-by-zero, we know that base and quote are both positive

```rust
/// The price of the base asset in terms of the quote.
#[derive(PartialEq, Eq, Debug, Clone, Copy)]
pub struct Price<Base, Quote> {
    price: PositiveDecimal,
    _base: PhantomData<Base>,
    _quote: PhantomData<Quote>,
}

impl<Base, Quote> Price<Base, Quote> {
    pub fn from_asset_ratios(
        base: PositiveAsset<Base>,
        quote: PositiveAsset<Quote>,
    ) -> Price<Base, Quote> {
        Price {
            price: quote.get_value() / base.get_value(),
            _base: PhantomData,
            _quote: PhantomData,
        }
    }
}
```

FP Block

Strongly Typed Messages

```rust
/// Name of an account owner
#[derive(
    serde::Serialize, serde::De
)]
pub struct Owner(pub String);
```

```rust
/// Messages that can be sent to the server
///
/// Note: using proper REST, gRPC, or Swagger would all be preferable.
/// Using this enum approach to demonstrate the power of serde for strong types.
#[derive(serde::Serialize, serde::Deserialize)]
#[serde(rename_all = "snake_case")]
pub enum ServerRequest {
    /// Get the overall system status.
    ///
    /// Returns: [StatusResp]
    Status {},
    /// Get the balance for the given owner.
    ///
    /// Returns: [BalanceResp]
    Balance { owner: Owner },
    /// Create new funds for a user.
    ///
    /// Returns: [MintFundsResp]
    MintFunds {
        recipient: Owner,
        usd_amount: UnsignedAsset<Usd>,
        euro_amount: UnsignedAsset<Euro>,
    },
    /// Convert dollars into euros
    ///
    /// Returns: [SellDollarsResp]
    SellDollars {
        trader: Owner,
        dollars: PositiveAsset<Usd>,
    },
```

```rust
#[derive(serde::Serialize, serde::Deserialize, Debug, Clone)]
pub struct StatusResp {
    /// Total amount of USD in both the pool and held by all users.
    pub total_usd: UnsignedAsset<Usd>,
    /// Total amount of EURO in both the pool and held by all users.
    pub total_euro: UnsignedAsset<Euro>,
    /// Price of a single USD in terms of EURO
    pub price_usd: Price<Usd, Euro>,
    /// Price of a single EURO in terms of USD
    pub price_euro: Price<Euro, Usd>,
}

#[derive(serde::Serialize, serde::Deserialize, Debug, Clone, Default)]
pub struct BalanceResp {
    pub usd: UnsignedAsset<Usd>,
    pub euro: UnsignedAsset<Euro>,
}

#[derive(serde::Serialize, serde::Deserialize, Debug, Clone)]
pub struct MintFundsResp {}

#[derive(serde::Serialize, serde::Deserialize, Debug, Clone)]
#[serde(rename_all = "snake_case")]
pub struct SellDollarsResp {
    pub euros_bought: PositiveAsset<Euro>,
}

#[derive(serde::Serialize, serde::Deserialize, Debug, Clone)]
#[serde(rename_all = "snake_case")]
pub struct SellEurosResp {
    pub dollars_bought: PositiveAsset<Usd>,
}
```

FP Block

Server Side Code

- Leveraging Axum for a web server
- Uses serde + JSON for serialization
- Data type prevent misusing values
- Serialization rules ensure the on-the-wire data is correct

Do we have time to check out the code itself?

https://github.com/snoyberg/rustikon-2025/blob/main/packages/server/src/main.rs

```rust
async fn status(&self) -> Result<StatusResp> {
    let mut total_usd = UnsignedAsset::zero(Usd);
    let mut total_euro = UnsignedAsset::zero(Euro);

    let guard = self.0.lock();

    for balance in guard.accounts.values() {
        total_usd += UnsignedAsset::new(Usd, balance.usd);
        total_euro += UnsignedAsset::new(Euro, balance.euro);
    }

    total_usd += guard.pool_usd.into_unsigned();
    total_euro += guard.pool_euro.into_unsigned();

    Ok(StatusResp {
        total_usd,
        total_euro,
        price_usd: Price::from_asset_ratios(guard.pool_usd, guard.pool_euro),
        price_euro: Price::from_asset_ratios(guard.pool_euro, guard.pool_usd),
    })
}
```

FP Block

I wrote a bug! Can you find it?

```
let pool_usd = guard.pool_usd;
let mut pool_euro = guard.pool_euro;
let owner = guard.accounts.entry(trader).or_default();
owner
    .usd
    .checked_sub_assign(euros.into_unsigned().into_decimal())?;

let k = pool_usd.into_unsigned().into_decimal() * pool_euro.into_unsigned().into_decimal();
```

- I promise I didn't do this on purpose
- But while testing, I found a bug in the code
- Can you see what the problem is?
- Can we prevent this from happening in the future?

FP Block

Needs moar types

- Problem: I was subtracting dollars and euros.
- Fix is easy: use the right field!

```
let mut pool_euro = guard.pool_euro;
let owner = guard.accounts.entry(trader).or_default();
owner
    .usd
    .euro
    .checked_sub_assign(euros.into_unsigned().into_decimal())?;
```

- Bigger problem: not enough types.
- Solution: use more types!

```
  #[derive(Default)]
  struct Balances {
-     usd: UnsignedDecimal,
-     euro: UnsignedDecimal,
+     usd: UnsignedAsset<Usd>,
+     euro: UnsignedAsset<Euro>,
  }
```

https://github.com/snoyberg/rustikon-2025/commit/956efd333e2865c23456a73167a3d5dec47fbcff

FP Block

Leptos Client

Frontend built using Leptos and leptos-query

I love the signal model, much nicer for me than React

But there are still some rough edges

Since it's Rust: reuse all the types

```rust
async fn perform_server_request<Resp: serde::de::DeserializeOwned>(
    req: ServerRequest,
) -> Result<Resp> {
    let req = serde_json::to_string(&req).map_err(Error::from_other_error)?;
    let res = reqwasm::http::Request::post("http://localhost:3001")
        .header("content-type", "application/json")
        .body(req)
        .send()
        .await
        .map_err(Error::from_other_error)?;
    if res.status() != 200 {
        return Err(Error::HttpRequestFailure {
            status: res.status(),
        });
    }
    res.json().await.map_err(Error::from_other_error)
}
```

FP Block

Component code

- Leptos has "components" like React
  - Slightly different in behavior
- Components can use a JSX-like syntax
- Data is all pure Rust
- All the normal display functions work in Leptos code
- Simple example: displaying wallet balances

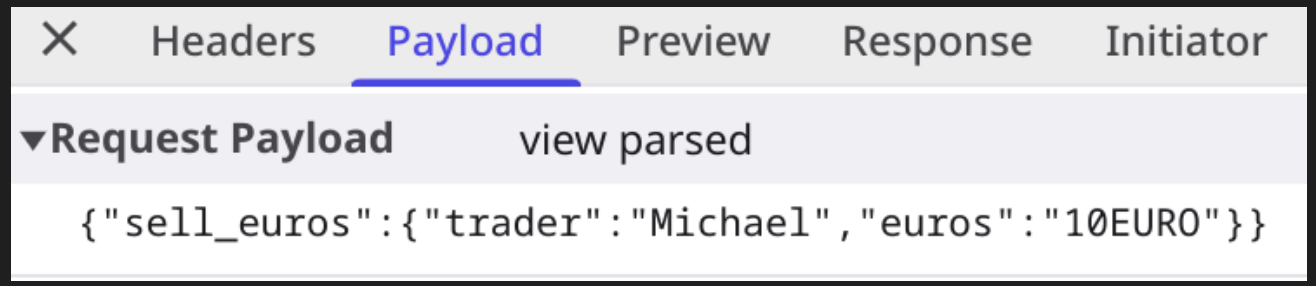
```
#[component]
pub(super) fn Owners() -> impl IntoView {
    let owners = query::owners().use_query(|| ());

    move || match owners.data.get() {
        None => view! { <i>Loading owner information</i> }.into_view(),
        Some(Err(e)) => {
            view! { <p class="error">Error loading owner information: {e.to_string()}</p> }
                .into_view()
        }
        Some(Ok(owners)) => view! {
            <For
                each=move || owners.clone()
                key=|owner| owner.owner.clone()
                children=move |balance| {
                    view! {
                        <p>
                            {balance.owner.to_string()}
                            " has "
                            {balance.dollars.to_string()}
                            " and "
                            {balance.euros.to_string()}
                        </p>
                    }
                }
            />
        },
    }
}
```

FP Block

Over the wire

- Data sent over the wire includes asset information
- Application code (server and client) never added that explicitly
- Just by using strong types that have been properly designed, we get extra guarantees at runtime!



FP Block

Thank you!

- End of the main content
- Happy to take questions now
- If there's time, we can load up the main app

Full source code: https://github.com/snoyberg/rustikon-2025

We're hiring Rust devs. Find me at the afterparty if you're interested.

FP Block