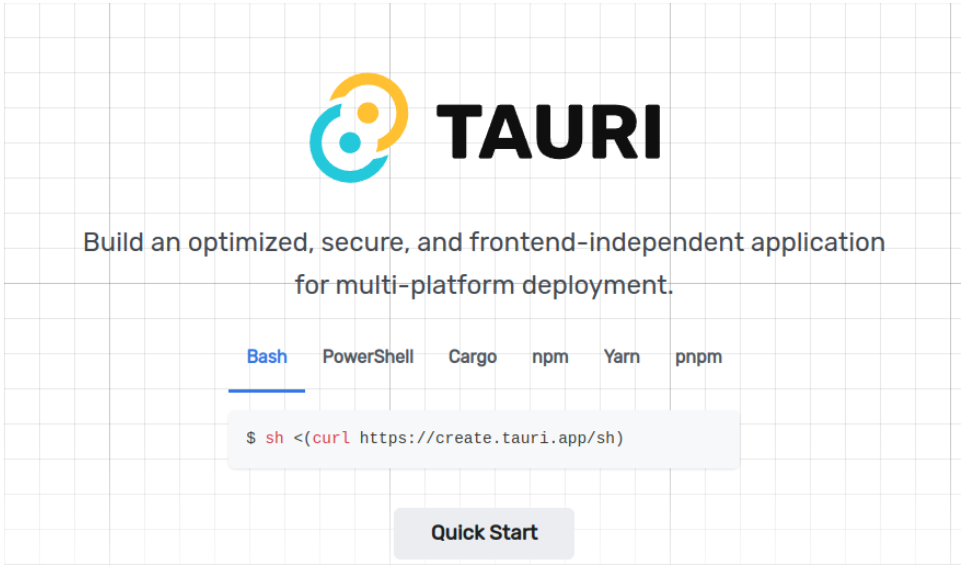
Rust

A brief intro

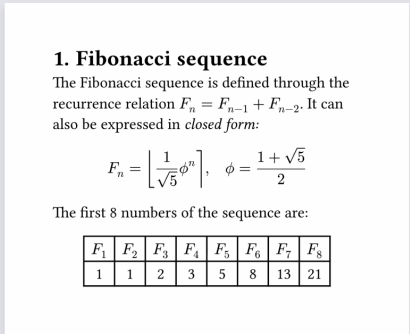
Miguel Palhas (@naps62) / Subvisual

Tauri



Typst

```
#set page(width: 10cm, height: auto)
#set heading(numbering: "1.")

= Fibonacci sequence
The Fibonacci sequence is defined through the
recurrence relation $F_n = F_(n-1) + F_(n-2)$.
It can also be expressed in _closed form:_

$ F_n = round(1 / sqrt(5) phi.alt^n), quad
  phi.alt = (1 + sqrt(5)) / 2 $

#let count = 8
#let nums = range(1, count + 1)
#let fib(n) = (
  if n <= 2 { 1 }
  else { fib(n - 1) + fib(n - 2) }
)

The first #count numbers of the sequence are:

#align(center, table(
  columns: count,
  ..nums.map(n => $F_#n$),
  ..nums.map(n => str(fib(n))),
))
```



1. Fibonacci sequence

The Fibonacci sequence is defined through the recurrence relation $F_n = F_{n-1} + F_{n-2}$. It can also be expressed in *closed form:*

$$F_n = \left\lfloor \frac{1}{\sqrt{5}} \phi^n \right\rceil, \quad \phi = \frac{1 + \sqrt{5}}{2}$$

The first 8 numbers of the sequence are:

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
1	1	2	3	5	8	13	21

Delta

redis-cli.c

```
static sds cliReadLine(int fd) {

    while(1) {
        char c;
        ssize_t ret;

        if (read(fd,&c,1) == -1) {
        ret = read(fd,&c,1);
        if (ret == -1) {
            sdsfree(line);
            return NULL;
        } else if (c == '\n') {
        } else if ((ret == 0) || (c == '\n')) {
            break;
        } else {
            line = sdscatlen(line,&c,1);
```

Just

```
alias b := build

host := `uname -a`

# build main
build:
    cc *.c -o main

# test everything
test-all: build
    ./test --all

# run a specific test
test TEST: build
    ./test --test {{TEST}}
```

```
: just -l                                   ~
Available recipes:
    build     # build main
    b         # alias for `build`
    test TEST # run a specific test
    test-all  # test everything
:                                           ~
```

DAVID HEINEMEIER HANSSON

September 7, 2023

Open source hooliganism and the TypeScript meltdown

We went from this

```
unresolved external symbol "void __cdecl
importStoredClients(class
std::basic_fstream<char,struct std::char_traits<char> > const
&,class
std::vector<class Client,class std::allocator<class Client> >
&)" (?
importStoredClients@@YAXABV?$basic_fstream@DU?
$char_traits@D@std@@@std@@AAV?
$vector@VClient@@V?$allocator@VClient@@@std@@@2@@Z)
```

… to this

```
1 + "2" == "12"
```

```
Undefined is not a function
```

Types of Types

Structural Typing (Typescript, OCaml, ...)

```
type Foo = { x: number, y: string };
type Bar = { x: number };

let x: Foo = { x: 1, y: "hello" };
```

Nominal Typing (Rust, C/C++, ...)

```rust
struct Foo { x: i32, y: String };
struct Bar { x: i32 };

fn main() {
  let x: Foo = Foo { x: 1, y: "hello".to_string() };

  // this would not compile
  // let y: Bar = x;
}
```

What makes Rust good?

1. Type inference

This is the same code

```
let list: Vec<u32> =
  vec![1u32, 2u32, 3u32].iter().map(|v: &u32| v +
1).collect::<Vec<u32>>()
```

```
let list =
  vec![1, 2, 3].iter().map(|v| v + 1).collect()
```

Type inference eliminates most noise.

Exceptions: function headers; ambiguity.

```
fn increment_and_dedup(v: Vec<u32>) -> HashSet<u32> {
  v.iter().map(|v| v + 1).collect()
}
```

2. Memory Safety

even in multi-threaded code

This fails to compile

```
fn main() {
  let x = 1;

  let r1 = &x;
  let r2 = &mut x;

  println!("{} {}", r1, r2);
}
```

Multi-threading type-safety

`trait Send`	safe to **send** to another thread
`trait Sync`	safe to **share** between threads

3. Compiler

Zero-cost abstractions

The ability to use high-level features without runtime cost.

Trade-off: compile-time complexity

“If it compiles, it works”

not to be taken literally

it’s how strongly typed programming **feels**

Making illegal states unrepresentable

Aim for compile-time safety, not runtime validations

- Type-drive development;
- Abuse `Option`, `Result`, and `enum`;
- Typestate pattern. (https://cliffle.com/blog/rust-typestate/)

Making illegal states unrepresentable

```
enum AccountState {
  Active { email: Email, active_at: DateTime },
  Inactive { email: Email },
  Banned { reason: String },
}

/// Newtype pattern
/// email regex can be enforced on constructor
/// runtime size is the same as String
type Email(String)
```

4. Tooling

```
cargo build
cargo run --package serve
cargo +nightly clippy
cargo fmt
cargo test
cargo build --target wasm32-unknown-unknown
cargo audit
bacon
```

Tips to get started

Don't get too Rust'y right away

- if you're writing `Foo<'a>`, you're gonna have a bad time;
- Abuse `clone()` instead of fighting the borrow checker;
- get v1 working, only then optimize.
- tooling will teach you along the way

Why NOT Rust?

bonus slides if I got here before the 20m mark

Compilation times?

- Incremental compilation is great(ish)
- Not quite instant-reload, but rather close
- Release builds are more painful
- You should `cargo check` instead of `cargo build`

Refactoring is a slog?

Closed • 858 total votes

643	Rafactoring in Rust is Easy
215	Refactoring in Rust is Hard

Voting closed 4 days ago

Best-Idiot • 11 days ago

When you change anything, a strong type system tells you what places still need to be fixed to adapt to the change

79 Reply Share ···

Sw429 • 10 days ago

Compare this to something like python, where you won't know that you missed changing something until you get a runtime exception for it, which sometimes might be missed until it's live in production.

31 Reply Share ···

Rust

A brief intro

Miguel Palhas (@naps62) / Subvisual