TS



RUSTIFY

From TypeScript to Rust
The Full Guide

For TS devs moving from "web" to serious backend

BIG PICTURE

How the transition usually unfolds

If you're reading this, you already know the ceiling. "TypeScript developer" is a useful label, but it's also the reason you get grouped into one of the largest talent buckets in software. The move to Rust is not starting over. It's moving your existing backend instincts into a rarer category.

For TypeScript engineers, the transition is usually: keep the type discipline, keep the async habits, keep the API design instincts, then add ownership, stronger guarantees, and a market position that gets read differently.

```
TYPESCRIPT -> RUST
│
├── WHAT ALREADY TRANSFERS
│   ├── types + generics          you already think in constraints
│   ├── async / await             the syntax and flow feel familiar
│   ├── API design                handlers, DTOs, validation, errors
│   └── backend instincts         auth, DB, queues, production tradeoffs
│
├── WHAT IS NEW
│   ├── ownership                 who owns what, and for how long
│   ├── borrowing                 references without garbage collection
│   ├── Result / Option habits    explicit happy path vs failure path
│   └── crate ecosystem           axum, tokio, sqlx, serde, tracing
│
├── WHAT CHANGES IN THE MARKET
│   ├── TypeScript                useful, but common
│   ├── Rust                      harder, but rarer
│   └── TS + Rust                 strong app experience + scarce backend skill
│
└── END STATE
    ├── ship a real Rust API
    ├── speak credibly with hiring managers
    └── move from "web dev" bucket to "serious backend" bucket
```

That is the logic of this guide: what transfers, what is genuinely new, why the market rewards the move, and why doing it sooner compounds faster than most TypeScript developers expect.

RUSTIFY

TABLE OF CONTENTS

From TypeScript to Rust

01 What transfers from TypeScript p.4

02 Ownership: the one thing that's new p.5

03 Error handling: goodbye try/catch p.6

04 Async Rust: familiar syntax, new rules p.7

05 Your first Axum API p.8

06 Is TypeScript still your differentiator? p.9

07 The market has a supply problem p.11

08 FAQ p.16

Targeting a Rust role? **Check if you're eligible for Rustify coaching →**

01 What transfers from TypeScri…
02 Ownership: the one thing that'…
03 Error handling: goodbye try/c…
04 Async Rust: familiar syntax, ne…
05 Your first Axum API
06 Is TypeScript still your differen…
07 The engineer you become afte…
08 The market has a supply probl…

01

What transfers from TypeScript

You already work in a typed, async-first language. More transfers than you think.

```
TypeScript                    Rust
──────────────────────────────────────────────────
async/await                →  async/await  (identical syntax)
T | null                   →  Option<T>    (no null exceptions, ever)
try/catch                  →  Result<T, E> (errors are return values)
Promise<T>                 →  Future<T>
interface / type           →  struct + trait
readonly                   →  let (immutable by default)
<T extends Foo>            →  <T: Foo>     (trait bounds)
T[]  /  Array<T>           →  Vec<T>
[T, U]                     →  (T, U)       (tuple)
Record<K, V>               →  HashMap<K, V>
never                      →  !
```

Enums with data — TypeScript's best feature, but better

```typescript
// TypeScript discriminated union — workable, verbose
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "rect"; width: number; height: number }
```

```rust
// Rust enum — first-class, exhaustively checked
enum Shape {
    Circle { radius: f64 },
    Rect { width: f64, height: f64 },
}
```

🎯 The compiler tells you when you've missed a case. Miss it in TypeScript and it's a runtime bug. Miss it in Rust and it won't compile.

01 What transfers from TypeScript
02 Ownership: the one thing tha…
03 Error handling: goodbye try/c…
04 Async Rust: familiar syntax, ne…
05 Your first Axum API
06 Is TypeScript still your differen…
07 The engineer you become afte…
08 The market has a supply probl…

02

Ownership: the one thing that's new

TypeScript has no equivalent for ownership. There's nothing to map it to — you have to build the model from scratch. One sentence: every value has exactly one owner. When the owner goes out of scope, the value is freed. No GC needed.

Move semantics — the shift that surprises everyone

In TypeScript, passing a value to a function passes a reference. The original stays. In Rust, passing a value moves it. The original is gone.

```
let name = String::from("Alice");
let greeting = make_greeting(name); // name moved here

println!("{}", name); // ✗ error: name was moved
```

The three solutions

```
// 1. Borrow — let the function look without owning
fn greet(name: &str) { println!("Hello, {}!", name); }
greet(&name); // name still here after

// 2. Clone — two independent copies
greet(name.clone());

// 3. Return ownership — function gives it back
let name = make_greeting(name);
```

⚠️ When the borrow checker rejects code, it's almost always pointing at a real design problem — not being pedantic. Read the error before reaching for `.clone()`.

01 What transfers from TypeScript
02 Ownership: the one thing that'...
03 Error handling: goodbye try/...
04 Async Rust: familiar syntax, ne...
05 Your first Axum API
06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...

03

Error handling: goodbye try/catch

Rust has no exceptions. Every function that can fail returns a `Result` type. Errors are regular values — not thrown, not swallowed, not forgotten. Every caller must decide what to do with them.

```
// TypeScript: can throw — caller may not know
function readFile(path: string): string { ... }

// Rust: explicit contract, enforced by compiler
fn read_file(path: &str) -> Result<String, io::Error> { ... }
```

The ? operator — ergonomic error propagation

The `?` operator propagates errors up the call stack automatically. Think of it as a try/catch that's honest about what it does.

```
fn process() -> Result<String, io::Error> {
    let content = read_file("data.txt")?; // error → return early
    let parsed  = parse(content)?;        // same
    Ok(parsed)
}
```

Common mistake to avoid

```
// ❌ .unwrap() panics in production — avoid in APIs
let data = read_file("data.txt").unwrap();

// ✅ propagate with ? instead
let data = read_file("data.txt")?;
```

🎯 Stop thinking of errors as exceptional. Think of them as a second return value. That shift changes everything about how you write Rust.

01 What transfers from TypeScript
02 Ownership: the one thing that'...
03 Error handling: goodbye try/c...
04 Async Rust: familiar syntax, n...
05 Your first Axum API
06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...

04

Async Rust: familiar syntax, new rules

The syntax is nearly identical. The semantics are stricter — in ways that catch real bugs.

```
// TypeScript
async function fetchUser(id: number): Promise<User> {
    return await db.find(id)
}

// Rust
async fn fetch_user(id: i64) -> Result<User, DbError> {
    let user = db.find(id).await?;
    Ok(user)
}
```

3 differences that matter

- **No built-in runtime** You pick Tokio. Add #[tokio::main] — then it disappears.
- **Futures are lazy** A Future does nothing until .await. A Promise starts immediately.
- **Send bounds** tokio::spawn requires Send. Holding a Mutex guard across .await = compile error. That's a race condition caught before production.

The crates you'll actually use

axum	HTTP server	(Express / Fastify)
tokio	async runtime	(Node.js event loop)
sqlx	SQL + type-safe	(Prisma / pg)
serde	serialization	(JSON.stringify / zod)
reqwest	HTTP client	(fetch / axios)
tracing	logging	(pino / winston)
thiserror	error types	—

🎯 Everything you need for a production backend API exists, is well-maintained, and compiles down to a single binary.

02 Ownership: the one thing that'...
03 Error handling: goodbye try/c...
04 Async Rust: familiar syntax, ne...
05 Your first Axum API
06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...
09 Is self-teaching Rust the best ...

05

Your first Axum API

Express developers find Axum familiar in structure and safer in every other way. Parameters are typed at the function signature level — no more `req.params.id` casting and hoping.

```
// Express
app.get('/users/:id', async (req, res) => {
    const user = await db.findUser(req.params.id) // string, hope it parses
    res.json(user)
})

// Axum — typed, validated before your handler runs
async fn get_user(
    Path(id): Path<i64>,
    State(db): State<Db>
) -> impl IntoResponse {
    let user = db.find_user(id).await?;
    Json(user)
}
```

Folder structure that works

```
src/
├── main.rs       router setup, DB pool init
├── errors.rs     AppError + IntoResponse  ← do this first
├── config.rs     env vars
├── routes/       one file per resource
├── models/       structs mapping to DB rows
└── services/     business logic, queries
```

🎯 Create errors.rs on day one. Every handler returns `Result` with your AppError type. Retrofitting error handling into an existing codebase is painful.

03 Error handling: goodbye try/c...
04 Async Rust: familiar syntax, ne...
05 Your first Axum API
06 Is TypeScript still your differe...
07 The engineer you become afte...
08 The market has a supply probl...
09 Is self-teaching Rust the best ...
10 The cost of "not now"

06

Is TypeScript still your differentiator?

In 2019, knowing TypeScript well meant something. It was new, adoption was uneven, and companies paid a premium for it. That gap has closed. TypeScript is now the default for serious frontend and a huge amount of backend work. Once a skill becomes default, it stops acting like leverage.

You'll always be good. You might always be replaceable.

There are millions of TypeScript developers. Senior ones. With years of React, Next.js, and Node. When a role opens up, that is your competition pool. Being strong in TypeScript is still valuable. It just is not rare anymore. You need to be rare in a different way.

The Node.js ceiling is real

Node handles most workloads fine. But when you hit scale, GC pauses, CPU-bound limits, or memory pressure, the answer isn't "better TypeScript." Companies at that point rewrite in Go or Rust. The engineers who already know Rust are the ones running those rewrites.

The market conflates you with "JS developers"

To most hiring managers, TypeScript and JavaScript still collapse into the same broad bucket. That bucket is full. Rust is a different conversation entirely. A TypeScript engineer who can also ship real Rust stops reading like "one more JS dev" and starts reading like a backend profile with a moat.

> 🎯 You're not deciding between TypeScript and Rust. You're deciding whether you want to stay in the big web bucket or move into a scarcer backend one.

04 Async Rust: familiar syntax, ne...
05 Your first Axum API
06 Is TypeScript still your differen...
07 The engineer you become aft...
08 The market has a supply probl...
09 Is self-teaching Rust the best ...
10 The cost of "not now"
11 "Rust is for systems devs, not ...

07

The engineer you become after Rust

The fear most TypeScript developers have is this: "I'll go back to being a junior." That is not what happens. You bring years of async patterns, type discipline, debugging habits, and API design experience into a language where many people are still trying to assemble those instincts.

You do not start over. You compound. That is the sentence to hold onto through the first hard weeks.

What TS devs report after 6 months in Rust

- They stop worrying about runtime errors. The compiler catches what used to surface in prod.
- They write less defensive code. No undefined checks, no try/catch boilerplate.
- They feel senior again, but now in a space where seniority is still scarce.
- Hiring managers treat them differently. A TS dev with Rust is unusual. A backend one is rarer still.

🎯 The transition is not from senior to beginner. It is from common senior profile to rarer senior profile.

Targeting a Rust role? Check if you're eligible for Rustify coaching →

05 Your first Axum API
06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...
09 Is self-teaching Rust the best ...
10 The cost of "not now"
11 "Rust is for systems devs, not ...
12 Why Rustify

08

The market has a supply problem

There are more Rust job openings than qualified developers. That is not a prediction. It is the current state. Companies are not waiting for the perfect Rust engineer to appear fully formed. They are hiring the best credible person available.

You do not need to be magical. You need to exist in a market where most engineers still have not shown up.

Where the demand actually is

- **Fintech & trading** Performance-critical backends where GC pauses are unacceptable
- **Infrastructure & DevTools** CLIs, build systems, platform tooling
- **B2B SaaS scale-ups** Rewriting bottlenecks from Node/Python as they hit scale
- **Web3 & embedded** Systems work that previously required C/C++

The hiring managers posting these roles are not finding what they need quickly. Many Rust openings sit open for months. When a backend engineer with strong TypeScript experience shows up and can talk Rust credibly, the conversation is different from the start.

> 🎯 The supply problem works in your favor. You don't need to outcompete hundreds of candidates. You need to be one of the few who showed up.

06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...
09 Is self-teaching Rust the best...
10 The cost of "not now"
11 "Rust is for systems devs, not ...
12 Why Rustify
13 FAQ

09

Is self-teaching Rust the best way?

When async Rust stopped making sense, who would you call?

You know async/await in TypeScript. Rust's async model looks familiar on the surface, then hits you with executors, pinning, and Send bounds. That's not a syntax problem. It's a mental model problem. Stack Overflow threads explain the error. They don't restructure how you think.

Do you know which crates you'd use for your first production API?

Which middleware pattern for auth in Axum? Which `sqlx` pattern for a transaction that touches 3 tables? This isn't in any tutorial. It's assembled over months, mostly wrong at first, unless someone shows you the short path.

How long before you'd feel confident shipping?

Most engineers who go alone give the same honest answer when pressed: probably a year, maybe more. That is not pessimism. It is what the path looks like when you are learning the language, the ecosystem, and the production patterns all at once without structure.

🎯 Self-teaching Rust is possible. The real question is whether your goal is to understand Rust eventually, or become legible for Rust roles within months.

06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...
09 Is self-teaching Rust the best ...
10 The cost of "not now"
11 "Rust is for systems devs, not ...
12 Why Rustify
13 FAQ

10

The cost of "not now"

You have a full-time job. Maybe a partner. Maybe kids. Real constraints, not excuses. This page is not about pretending otherwise.

But "no time" and "this isn't a priority right now" are different things. And knowing which one is true changes everything.

What specifically needs to change before you'd have time?

A project wraps up. A release ships. Things calm down. When did they last calm down, and what did you do with that time? The engineers shipping Rust today usually did not wait for a magical quiet period. They decided this was a category shift worth prioritizing before life became convenient.

What would you have today if you'd started 6 months ago?

A deployed Axum service. A credible Rust proof point. A different set of conversations with hiring managers. Six months from now, this question has the same answer unless something changes now.

Is it time, or is it priority?

If you genuinely have no bandwidth right now, that's a real answer. But if the honest answer is "I'm not sure it's worth it": that's a different question. And it's the one this guide is trying to help you answer.

> 🎯 The engineers who say "I'll do it when things calm down" are still waiting. The ones who shipped are the ones who decided before things calmed down.

06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...
09 Is self-teaching Rust the best ...
10 The cost of "not now"
11 "Rust is for systems devs, not...
12 Why Rustify
13 FAQ

11

"Rust is for systems devs, not web devs"

That was true in 2016. Today, many new Rust developers come from TypeScript, Python, and JavaScript. The systems crowd opened the door. The backend web world walked through it.

Your background is closer than you think

You already write async code every day. You reason about types, interface contracts, and error handling. You've shipped APIs under production load. Most early Rust devs from the systems world are still learning that. You arrive with it.

You'll be ahead, not behind

A TypeScript developer who transitions brings modern async instincts, type discipline, and API design experience. What you do not yet have is ownership. That is one genuinely new model, not ten years of C++ baggage.

The question isn't where you come from

The Rust community doesn't ask about your background. It asks what you can build. Two years from now, you'll be the senior Rust dev that new people look to for guidance. The impostor is the one who never starts.

🎯 You are not crossing from a fake background into a real one. You are moving from one engineering bucket into another.

RUSTIFY

06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...
09 Is self-teaching Rust the best ...
10 The cost of "not now"
11 "Rust is for systems devs, not ...
12 Why Rustify
13 FAQ

SUCCESS STORIES

TS devs who made the switch



Ugo Tiberto

Rust Engineer · Fullstack

Ugo started with JavaScript 3 years ago, then decided to push further. He joined the bootcamp because learning Rust basics alone is easy, but building real apps with Axum, Leptos, and SQLx requires structure. Two weeks after finishing, he had a production Leptos app deployed.

Eddy Kamto

Software Engineer · Rust / TS

Eddy is a fullstack engineer with 5 years of TypeScript. He joined after seeing Rust UI, a Shadcn-like component library built entirely in Rust. He was looking for a structured path through a language with a steep curve and a vast ecosystem.





Tiago Afonso

Fullstack Developer

Tiago studied computer science in India and graduated in 2024. He first encountered Rust through Bitcoin infrastructure work. He joined the bootcamp after seeing a LinkedIn post about using Rust for full-stack web development, not just systems programming.

06 Is TypeScript still your differen...
07 The engineer you become afte...
08 The market has a supply probl...
09 Is self-teaching Rust the best ...
10 The cost of "not now"
11 "Rust is for systems devs, not ...
12 Why Rustify
13 FAQ

FAQ

Q1. I got stuck on the borrow checker for days. Is that normal?

Yes — it's almost universal. The borrow checker enforces a model TypeScript never asked you to think about. Getting stuck is the learning. It clicks in week 2–3 of actual practice, not passive reading.

Q2. Can't I learn this from free content?

The Rust Book is excellent. But most engineers who go alone stall after it — no project pulling them forward. The Book teaches the language. Building an API with auth, a database, and error handling teaches you how to ship.

Q3. Can I use Rust for full-stack development?

Yes. Leptos is a full-stack Rust framework — SSR, reactivity, shared types between frontend and backend. No TypeScript layer. One language, end-to-end. TypeScript devs adapt fast because the component model is familiar.

Q4. Is Node.js good enough? Why bother?

For most workloads, yes. But when companies hit CPU, memory, or latency ceilings, they stop asking for better Node and start rewriting. Rust is where those rewrites go. The engineers running them already knew Rust.

Q5. How long until I'm actually productive?

For TypeScript developers: 4–6 weeks to ship something real if you're focused. Ownership is the bottleneck. After it clicks, your TS instincts do the rest — async, types, API design all transfer directly.

Q6. Should I learn Go instead? It's easier.

Go is easier to learn and easier to find. The supply of Go devs grows fast. Rust requires more investment but the supply of qualified engineers stays structurally low. Speed of entry vs depth of moat — pick one.

Targeting a Rust role? **Check if you're eligible for Rustify coaching →**



Ready to leave the "JS dev" bucket?

Your TypeScript experience is not a sunk cost.
It is the foundation of a rarer backend profile.
Check if you qualify.

Check my eligibility →