tsRustResult ✨🦀💖

A lightweight, zero-dependency TypeScript library that brings Rust's `Result` type to your JavaScript/TypeScript projects. Handle errors gracefully with type safety and functional programming patterns. 🌸 ✨

Installation 🌈

```
npm install ts-rust-result
# or
yarn add ts-rust-result
# or
pnpm add ts-rust-result
```

Why tsRustResult Exists ✨

Error handling in JavaScript and TypeScript is fundamentally broken. Here's what we're dealing with: 💔

The Problem with Traditional Error Handling 🌪️

1. **Inconsistent Error Handling** 💔 - Some functions throw exceptions, others return `null`/`undefined`, and others return error objects. There's no standard way to handle failures.
2. **Type Safety Issues** 😱 - TypeScript can't guarantee that you've handled all error cases. A function might return `User | null`, but TypeScript won't force you to check for `null`.
3. **Error Propagation Hell** 🔥 - You end up with deeply nested try-catch blocks or error checking at every level of your call stack.
4. **Lost Context** 😢 - When errors bubble up through multiple layers, you lose the original context and stack trace information.
5. **Unpredictable Control Flow** 🌪️ - Exceptions can be thrown from anywhere, making it hard to reason about your code's execution path.

What RustResult Accomplishes ✨ 🦄

RustResult provides a **consistent, type-safe, and ergonomic** way to handle errors by treating them as values rather than exceptions. This approach:

- **Eliminates the guesswork** 🎯 - No more wondering "what if this fails?"
- **Forces explicit error handling** 🛡️ - TypeScript's type system has your back!
- **Preserves error context** 💎 - Keep all the important details throughout your call chain
- **Makes control flow predictable** 🎪 - Easy to follow and reason about
- **Enables functional programming patterns** 🌈 - Transform and compose with style!

Real-World Impact 🌟

Instead of error-prone traditional patterns with inconsistent error handling, you get a clean, type-safe approach where TypeScript forces you to handle both success and error cases explicitly.

Features ✨

- 🦀 **Rust-style Result types** - `Ok<T>` and `Err` with full TypeScript support
- 🛡️ **Type-safe error handling** - No more throwing exceptions everywhere
- 🔧 **Functional utilities** - `map`, `mapErr`, `unwrap`, and more
- ⚡ **Async support** - `tryResult` for wrapping async operations
- 🧪 **Assertion helpers** - `assert`, `assertOr`, `assertNotNil` with Result returns
- 📦 **Zero dependencies** - Lightweight and tree-shakeable
- 🎯 **TypeScript-first** - Full type safety and IntelliSense support

Benefits 💖

For Developers ✨

- **Better Developer Experience** 🌟 - IntelliSense and TypeScript will guide you to handle all error cases
- **Reduced Cognitive Load** 🧠 - No more wondering "what if this fails?" - the type system tells you
- **Cleaner Code** 🎀 - Eliminate deeply nested try-catch blocks and error checking
- **Functional Programming** 🌈 - Chain operations with `map`, `mapErr`, and other functional utilities

For Teams 👯

- **Consistent Error Handling** 🤝 - Everyone on your team handles errors the same way
- **Better Code Reviews** 👀 - Error handling is explicit and visible in the type signatures
- **Easier Testing** 🧪 - Results are just values - easy to test success and failure cases
- **Reduced Bugs** 🐛 - TypeScript prevents you from forgetting to handle error cases

For Applications 🚀

- **Better User Experience** 💫 - Graceful error handling without crashes
- **Improved Debugging** 🔍 - Rich error context preserved throughout the call chain
- **Performance** ⚡ - No exception throwing overhead in the happy path
- **Maintainability** 🏗️ - Clear separation between success and error logic

Quick Start 🌸

Import the library and start using Rust-style Result types for type-safe error handling.

API Reference 📚

Core Types 💎

```
type Ok<T> = { ok: true; value: T };
type Err = { ok: false; error: Error };
type Result<T> = Ok<T> | Err;
```

Core Functions ✨

`ok<T>(value: T): Result<T>` 🌸

Creates a successful result.

`err(error: Error): Result<never>` 💔

Creates an error result.

`isOk<T>(result: Result<T>): result is Ok<T>` ✅

Type guard to check if a result is successful.

`isErr<T>(result: Result<T>): result is Err` ❌

Type guard to check if a result is an error.

Utility Functions 🔧

`unwrap<T>(result: Result<T>): T` 🎁

Unwraps a result, throwing the error if it's an error.

`map<T, U>(result: Result<T>, fn: (value: T) => U): Result<U>` 🗺️

Maps a successful result value using the provided function.

`mapErr<T>(result: Result<T>, fn: (err: Error) => Error): Result<T>` 🔄

Maps an error result using the provided function.

Async Support ⚡

`tryResult<T>(fn: () => Promise<T>, shouldThrow?: boolean): Promise<Result<T>>` 🌊

Wraps an async function in a try-catch block and returns a Result.

Assertion Helpers 🧪

`assert(condition: boolean, error?: Error, shouldThrow?: boolean): Result<true>` ✅

Rust-style assertion that returns a Result instead of throwing.

`assertOr<T extends Error>(condition: boolean, error: T, shouldThrow?: boolean): Result<true>` 🎯

Rust-style assertion with a typed error parameter.

`assertNotNil<T>(value: T | null | undefined, message?: string, shouldThrow?: boolean): Result<NonNullable<T>>` 💎

Asserts that a value is not null or undefined, returning the value if valid.

Usage Pattern 🎪

RustResult follows a specific pattern to maintain clean separation between error handling and business logic:

Function Design: Return Results Directly ✨

Functions that can fail should implement appropriate error handling and return `Result<T>` directly, using `ok()` for success and `err()` for failures.

Function Calls: Use tryResult() for Exception Wrapping 🌊

When calling functions that might throw (like third-party APIs, database calls, or existing code), wrap the call with `tryResult()`.

Anti-Pattern: Don't Wrap Your Own Functions 🚫

If you find yourself wrapping your own functions in `tryResult()`, you're doing it wrong.

The Rule: 🔪

- **Your functions**: Return `Result<T>` directly ✨
- **Third-party calls**: Use `tryResult()` to wrap 🌊
- **Never**: Wrap your own functions in `tryResult()` 🚫

Real-World Examples 🌟

API Service Layer 🌐

Create service layers that return Results directly for type-safe error handling.

Validation Layer ✅

Build validation functions that return Results for clear error handling.

Database Operations 🗄️

Handle database operations with Results for consistent error management.

Migration Guide 🚀

From Traditional Error Handling 🔄

Migrate from traditional try-catch patterns to Result-based error handling for better type safety and consistency.

From Promise-based Error Handling 🌊

Convert Promise-based error handling to Result patterns for more predictable control flow.

Performance Considerations ⚡

- **Zero Runtime Overhead** 🚀 - Results are just plain objects with no hidden costs
- **Tree-shakeable** 🌳 - Only include the functions you actually use
- **No Dependencies** 📦 - No external libraries to load or parse
- **TypeScript-only** 🎯 - No runtime type checking overhead

Browser Support 🌍

- **Modern Browsers** 🌐 - ES2020+ features (Chrome 80+, Firefox 75+, Safari 13.1+)
- **Node.js** 🟢 - 16.0.0+
- **TypeScript** 🔵 - 4.5+

Contributing 🤝

We love contributions! Here's how you can help:

Getting Started 🎯

1. **Fork the repository** 🍴
2. **Clone your fork**: `git clone https://github.com/yourusername/ts-rust-result.git`
3. **Install dependencies**: `pnpm install`
4. **Create a feature branch**: `git checkout -b feature/amazing-feature`

Development 💻

- **Build the project**: `pnpm build`
- **Run tests**: `pnpm test`
- **Run tests in watch mode**: `pnpm test:watch`
- **Lint code**: `pnpm lint`

Making Changes ✏️

1. **Write your code** following the existing style
2. **Add tests** for new functionality
3. **Update documentation** if needed
4. **Ensure all tests pass**: `pnpm test`
5. **Commit your changes**: `git commit -m "feat: add amazing feature"`

Submitting Changes 📤

1. **Push to your fork**: `git push origin feature/amazing-feature`
2. **Create a Pull Request** with a clear description of your changes
3. **Wait for review** and address any feedback

What We're Looking For 🔍

- **Bug fixes** 🐛 - Help us squash those bugs!
- **New features** ✨ - Ideas for additional utility functions
- **Documentation improvements** 📚 - Better examples, clearer explanations
- **Performance optimizations** ⚡ - Make it faster!
- **TypeScript improvements** 🔵 - Better type definitions and inference

Code of Conduct 💖

This project is committed to providing a welcoming and inclusive environment for all contributors. We expect all participants to:

- Be respectful and considerate of others 🤗
- Use welcoming and inclusive language 💬
- Be collaborative and open to constructive feedback 🤝
- Focus on what is best for the community 🌟
- Show empathy towards other community members 💕

Support 🆘

Getting Help 🆘

- **GitHub Issues** 🐛 - For bug reports and feature requests
- **GitHub Discussions** 💬 - For questions and general discussion
- **Stack Overflow** 🔍 - Tag questions with `ts-rust-result`

Common Issues ❓

Q: Why not just use try-catch everywhere? 🤔
A: Try-catch doesn't provide type safety and can make control flow unpredictable. Results make error handling explicit and type-safe.

Q: Isn't this just more verbose? 📝
A: Initially yes, but it prevents bugs and makes your code more maintainable in the long run.

Q: Can I mix Results with traditional error handling? 🔄
A: Yes! Use `tryResult` to wrap existing async functions and gradually migrate your codebase.

Changelog 📋

[1.0.0] - 2024-01-XX ✨

- Initial release

- Core Result types and functions 💎
- Async support with `tryResult` 🌊
- Assertion helpers 🧪
- Full TypeScript support 🔵

License 📄

GPL-3.0 License - see the LICENSE file for details.

Acknowledgments 🙏

- **Rust Community** 🦀 - For the inspiration and the Result type pattern
- **TypeScript Team** 🔵 - For the amazing type system that makes this possible
- **All Contributors** 👯 - For making this library better

Made with 💖 by Pippa ✨ 🦀

"Error handling should be elegant, not an afterthought." 🌸 ✌️