Error Handling

with less pain*

* hopefully 🤞

What will we talk about?

- Motivation
- Solution
- Patterns
- Benefits
- Downsides

Motivation

- applications start with nice logic
- gets complex with need of error handling
- e.g. error recovery, logging, reporting ... etc

Motivation

```
const registerUser = async (input: CreateUserInput, db: Db): Promise<void> => {
  const trimmedEmail = input.email.trim()
  const domain = trimmedEmail.split("@")[1]
  const company = await db.companies.find({ domain })
  const user: User = { ...input, email: trimmedEmail, company }
  await db.users.create(user)
}
```

nice and simple!

Motivation

then error handling happens:

```
const registerUser = async (input: CreateUserInput, db: Db): Promise<void> => {
  const age = input.dateOfBirth.diff(new Date(), "days")
  if (age < 18) throw new Error("Must be older than 18 years")

  const domain = input.email.split("@")[1]
  if (!domain) throw new Error("Email is invalid")
  if (domain.includes("microsoft") || domain.includes("outlook"))
    throw new Error(`Cannot register as ${domain} is banned`)

  try {
    const company = await db.companies.find({ domain })
    const user: User = { ...input, age, company }
    await db.users.create(user)
  } catch (error) {
    if (error.message.includes("domain not found"))
      throw new Error("Company not found")
  }
}
```

Motivation

- Can we tell which errors are happening without reading source code?
- Can we enforce that these errors are handled by devs?
- Can we handle these errors in a way that it would not break if the error copy changes?
- Can we reuse the function? (e.g. use in another function that allows admins to bypassing some of these errors)
- Can we list all erros so that we could display them to users in one go?

Motivation

Trigger warning:
some viewers might find the following function triggering.

```
const placeOrder = (user, location, items, date, time, payment): void =>
  😱
```

Motivation

What about when failures are not modelled as errors?

```
const pricePerHead = totalPrice / numberOfPeople // could return Infinity
```

```
parseInt("yo") // NaN
```

Solution

Treat unhappy paths the same way we treat happy ones!

EQUALITY

Solution

```
const validateDateOfBirth = (date: Date): boolean => {
  if (date.isAfter(new Date()))
    throw new Error("Date of birth cannot be in future")
  const age = date.diff(new Date(), "years")
  return age >= 18
}
```

Solution

Let's model all paths in the return type:

```ts
type ValidateDateOfBirthError =
  | { type: "date_in_future" }
  | { type: "too_young"; age: number }

type ValidateDateOfBirthResult =
  | { type: "success"; age: number }
  | { type: "error"; reason: ValidateDateOfBirthError }

const validateDateOfBirth = (date: Date): ValidateDateOfBirthResult => {
  if (date.isAfter(new Date()))
    return { type: "error", reason: { type: "date_in_future" } }
  const age = date.diff(new Date(), "years")
  return age >= 18
    ? { type: "success", age }
    : { type: "error", reason: { type: "too_young", age } }
}
```

Solution

Less repetition please:

```
type Result<S, E> = { type: "success"; value: S } | { type: "error"; reason: E }
```

```
type ValidateDateOfBirthError =
  | { type: "date_in_future" }
  | { type: "too_young"; age: number }
const validateDateOfBirth = (
  date: Date,
): Result<number, ValidateDateOfBirthError> => {
  if (date.isAfter(new Date()))
    return { type: "error", reason: { type: "date_in_future" } }
  const age = date.diff(new Date(), "years")
  return age >= 18
    ? { type: "success", age }
    : { type: "error", reason: { type: "too_young", age } }
}
```

Solution

Even less repetition:

```
const ok = <S>(value: S): Result<S, never> => ({ type: "success", value })
const err = <E>(reason: E): Result<never, E> => ({ type: "error", reason })
```

```
type ValidateDateOfBirthError =
  | { type: "date_in_future" }
  | { type: "too_young"; age: number }
const validateDateOfBirth = (
  date: Date,
): Result<number, ValidateDateOfBirthError> => {
  if (date.isAfter(new Date())) return err({ type: "date_in_future" })
  const age = date.diff(new Date(), "years")
  return age >= 18 ? ok(age) : err({ type: "too_young", age })
}
```

Solution

Don't want to reinvent the wheel?

```
npm i neverthrow
```

```ts
import { ok, err, Result } from "neverthrow"

type ValidateDateOfBirthError =
  | { type: "date_in_future" }
  | { type: "too_young"; age: number }
const validateDateOfBirth = (
  date: Date,
): Result<number, ValidateDateOfBirthError> => {
  if (date.isAfter(new Date())) return err({ type: "date_in_future" })
  const age = date.diff(new Date(), "years")
  return age >= 18 ? ok(age) : err({ type: "too_young", age })
}
```

Patterns

```
const readFile = (name: string): Result<string, ReadFileError> => ...
```

```
readFile("secret.txt") // Result<string, ReadFileError>
  .map((content) => content.length) // Result<number, ReadFileError>
```

Patterns

```
const decrypt = (content: string): Result<string, DecryptError> => ...
```

```
readFile("secret.txt") // Result<string, ReadFileError>
  .andThen(decrypt) // Result<string, ReadFileError | DecryptError>
```

Patterns

```
readFile("secret.txt") // Result<string, ReadFileError>
  .match(
    (content) => sendSecretToPartnerService(content),
    (failure) => reportErrorToDatadog(failure),
  ) // void
```

```
createOrderResult.match(
  (order) => <h1>Your order ID is {order.id}</h1>,
  (failure) => (
    <div>Could not place order. Reason: {formatFailure(failure)}</div>
  ),
)
```

Patterns

```
const validateAge = (age: number) => ... // Result<number, ValidateAgeError>
const validateEmail = (email: string) => ... // Result<string, ValidateEmailError>
const findCompanyByEmail = (email: string) => ... // Result<Company, FindCompanyByEmailError>
```

```
Result.combine([
  validateAge(user.age),
  validateEmail(user.email),
  findCompanyByEmail(user.email),
])
// Result<
//   [age: number, email: string, company: Company],
//   ValidateAge | ValidateEmailError | FindCompanyByEmailError
// >
```

Patterns

What about Promises?

```
import { ResultAsync } from "neverthrow"

type GetUserError = { type: "db_error"; cause: PrismaError }
const getUser = (id: number): ResultAsync<User, GetUserError> =>
  ResultAsync.fromPromise(
    db.users.find(id), //
    (cause) => ({ type: "db_error", cause }),
  )
```

```
getUser(id)
  .map((user) => user.email)
  .andThen(sendEmail)
  .match(
    (email) => console.log("Email sent to " + email.to),
    (error) => console.error("Failed to send email: " + JSON.stringify(error)),
  )
```

Patterns

```ts
const validateAge = (age: number) => ... // Result<number, ValidateAgeError>
const validateEmail = (email: string) => ... // Result<string, ValidateEmailError>
const findCompanyByEmail = (email: string) => ... // Result<Company, FindCompanyByEmailError>

type CreateUserError = ValidateAgeError | ValidateEmailError | FindCompanyByEmailError
const createUser = (input: CreateUserInput): Result<User, CreateUserError> =>
  Result.combine([validateAge(input.age), validateEmail(input.email), findCompanyByEmail(input.email)])
    .map(([age, email, company]) => ({ name: input.name, age, email, company }))
```

```ts
import { match } from "ts-pattern"

const formatCreateUserError = (error: CreateUserError): string =>
  match(error)
    .with(
      { type: "too_young" },
      ({ age }) => `You're only ${age}. Come back when you're 18`,
    )
    ...
    .exhaustive()
```

Benefits

- discoverability
- readability
- reusability
- enforcement by compiler

Downsides

- learning curve
- return early performance

The End

Demo