Result Types

Safe and Sound Error Handling

Introduction

"Happy Path Blindness"

What could go wrong?

- 🕵️ The data doesn't exist
- 🔥 The network is down
- 💥 The server returns an error
- 🤷 The server returns a different shape

```
 1   const getData = async <T>(from: string) ⇒ {
 2     const url = `/api/data?from=${from}`;
 3     const res = await fetch(url);
 4     if (!res.ok) {
 5       throw new Error(res.statusText);
 6     }
 7     return (await res.json()) as T;
 8   };
 9
10   const main = async () ⇒ {
11     const data = await getData<DataT[]>("2024");
12   };
```

"Every function you write has two sets of inputs and outputs"

~ Kris Jenkins
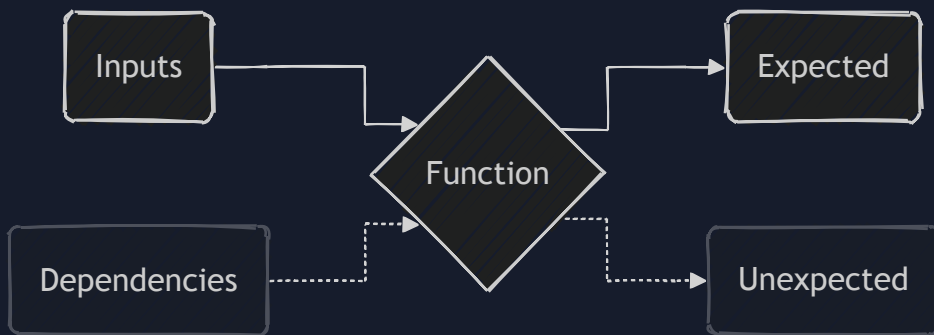
How do other languages do it?

```go
func divide(a, b int) (int, error) {
    if b == 0 {
        return 0, errors.New("division by zero")
    }
    return a / b, nil
}

func main() {
    result, err := divide(10, 0)
    if err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println("Result:", result)
    }
}
```

Haskell

```haskell
divide :: Int → Int → Either String Int
divide a b = if b == 0
  then Left "division by zero"
  else Right (a `div` b)

main :: IO ()
main = do
  let result = divide 10 0
  case result of
    Left err → putStrLn $ "Error: " ++ err
    Right val → putStrLn $ "Result: " ++ show val
```

Rust

```rust
fn divide(a: i32, b: i32) → Result<i32, String> {
  if b == 0 {
    Err("division by zero".to_string())
  } else {
    Ok(a / b)
  }
}

fn main() {
  match divide(10, 0) {
      Ok(result) ⇒ println!("Result: {}", result),
      Err(e) ⇒ println!("Error: {}", e),
  }
}
```

What is a Result Type?

tryCatch Block

```typescript
const getData = async <T>(from: string) ⇒ {
  try {
    const url = `/api/data?from=${from}`;
    const res = await fetch(url);
    if (!res.ok) {
      throw new Error(res.statusText);
    }
    return (await res.json()) as T;
  } catch (error) {
    console.error(error);
    return [] as T;
  }
};

const main = async () ⇒ {
  const data = await getData<DataT[]>("2024");
};
```

Discriminate Union

```typescript
const getData = async <T>(from: string) ⇒ {
  const url = `/api/data?from=${from}`;
  const res = await fetch(url);
  if (!res.ok) {
    throw new Error(res.statusText);
  }
  return (await res.json()) as T;
};

const main = async () ⇒ {
  const result = await tryCatch(
    getData<DataT[]>("2024")
  );
  if (result.error) {
    console.log("Unable to get data");
    return;
  }
  const data = result.data;
};
```

```typescript
type Success<T> = {
  data: T;
  error: null;
};

type Failure<E> = {
  data: null;
  error: E;
};

type Result<T, E = Error> =
  | Success<T>
  | Failure<E>;

const tryCatch = async <T, E = Error>(
  promise: Promise<T>
): Promise<Result<T, E>> ⇒ {
  try {
    const data = await promise;
    return { data, error: null };
  } catch (error) {
    return { data: null, error: error as E };
  }
};
```

Either

```
1   const getData = async <T, E = Error>(
2     from: string
3   ): Promise<Either<E, T>> ⇒ {
4     try {
5       const url = `/api/data?from=${from}`;
6       const res = await fetch(url);
7       if (!res.ok) {
8         return left(new Error(res.statusText) as E);
9       }
10      const data = (await res.json()) as T;
11      return right(data);
12    } catch (error) {
13      return left(
14        new Error("Unable to fetch data") as E
15      );
16    }
17  };
```

```
1   type Left<T> = {
2     _tag: "Left";
3     left: T;
4   };
5
6   type Right<T> = {
7     _tag: "Right";
8     right: T;
9   };
10
11  type Either<L, R> = Left<L> | Right<R>;
12
13  const left = <L, R>(left: L): Either<L, R> ⇒ {
14    return {
15      _tag: "Left",
16      left,
17    };
18  };
19  const right = <L, R>(right: R): Either<L, R> ⇒ {
20    return {
21      _tag: "Right",
22      right,
23    };
24  };
```

Feature	*try/catch Block*	*Discriminate Union*	*Either Type*
Pattern Style	Imperative	Declarative	Functional
Error Handling	Implicit (can forget to catch)	Explicit and enforced by types	Explicit and functional
Type Safety	❌ Inconsistent return types	✅ Strongly typed	✅ Strongly typed
Composability	❌ Difficult to compose	✅ Easy to chain and compose	✅ Excellent for functional composition
Pattern Matching	❌ Not supported	✅ Via discriminated union	✅ With `_tag` matching
Example Use	Small scripts, fallback logic	General-purpose, app-safe async calls	Functional pipelines, FP-heavy codebases

effect

```
 1  const getData = async (
 2    from: string
 3  ): Promise<
 4    Either.Either<
 5      ReadonlyArray<typeof Datum.Type>,
 6      DataFailure
 7    >
 8  > ⇒ {
 9    try {
10      const url = `/api/data?from=${from}`;
11      const res = await fetch(url);
12      if (!res.ok) {
13        return Either.left(
14          new BadServerResponse({
15            message: `Bad server response: ${res.statusText}`,
16          })
17        );
18      }
19      const data = await res.json();
20      const parseResult =
21        Schema.decodeUnknownEither(
22          Schema.Array(Datum)
```
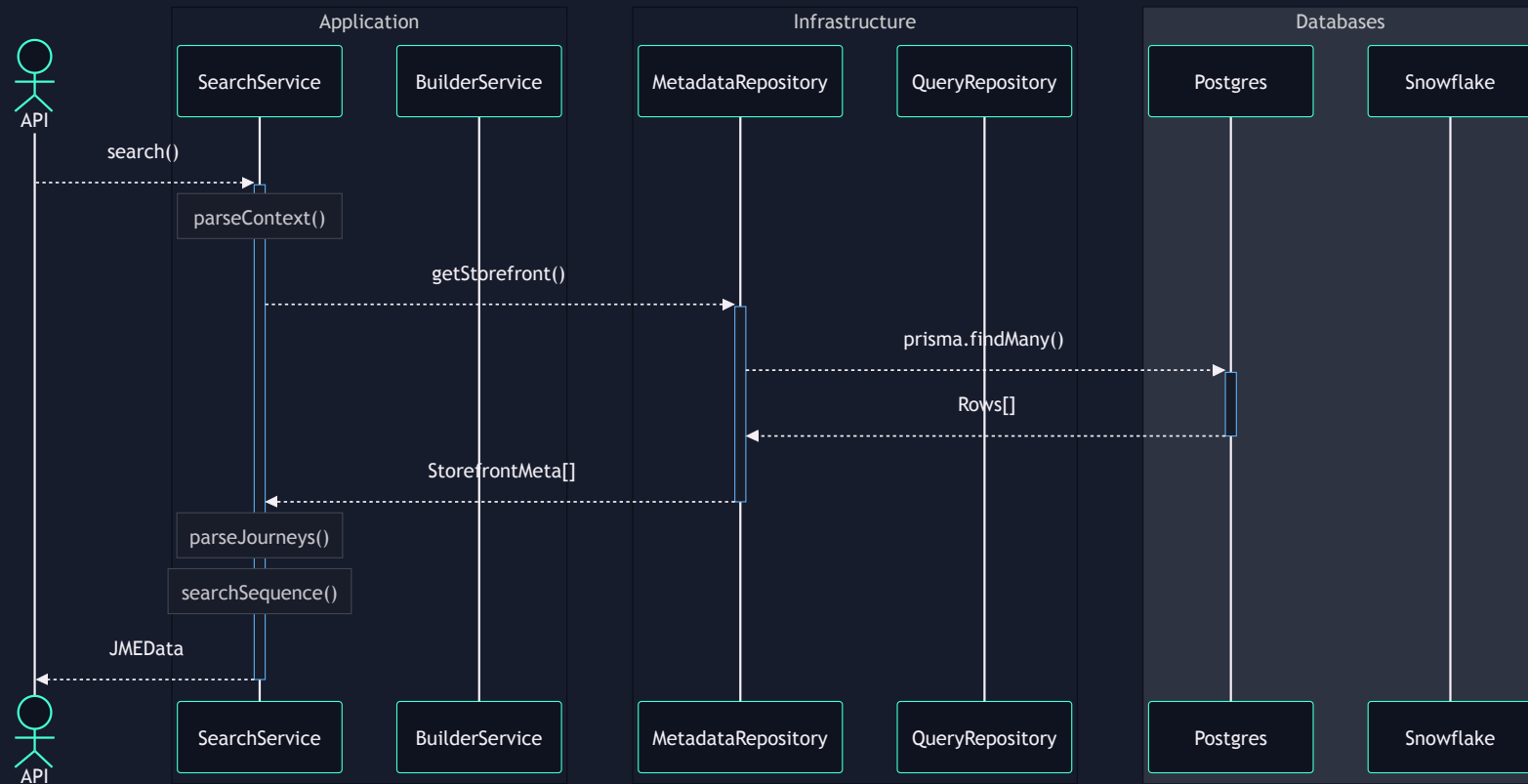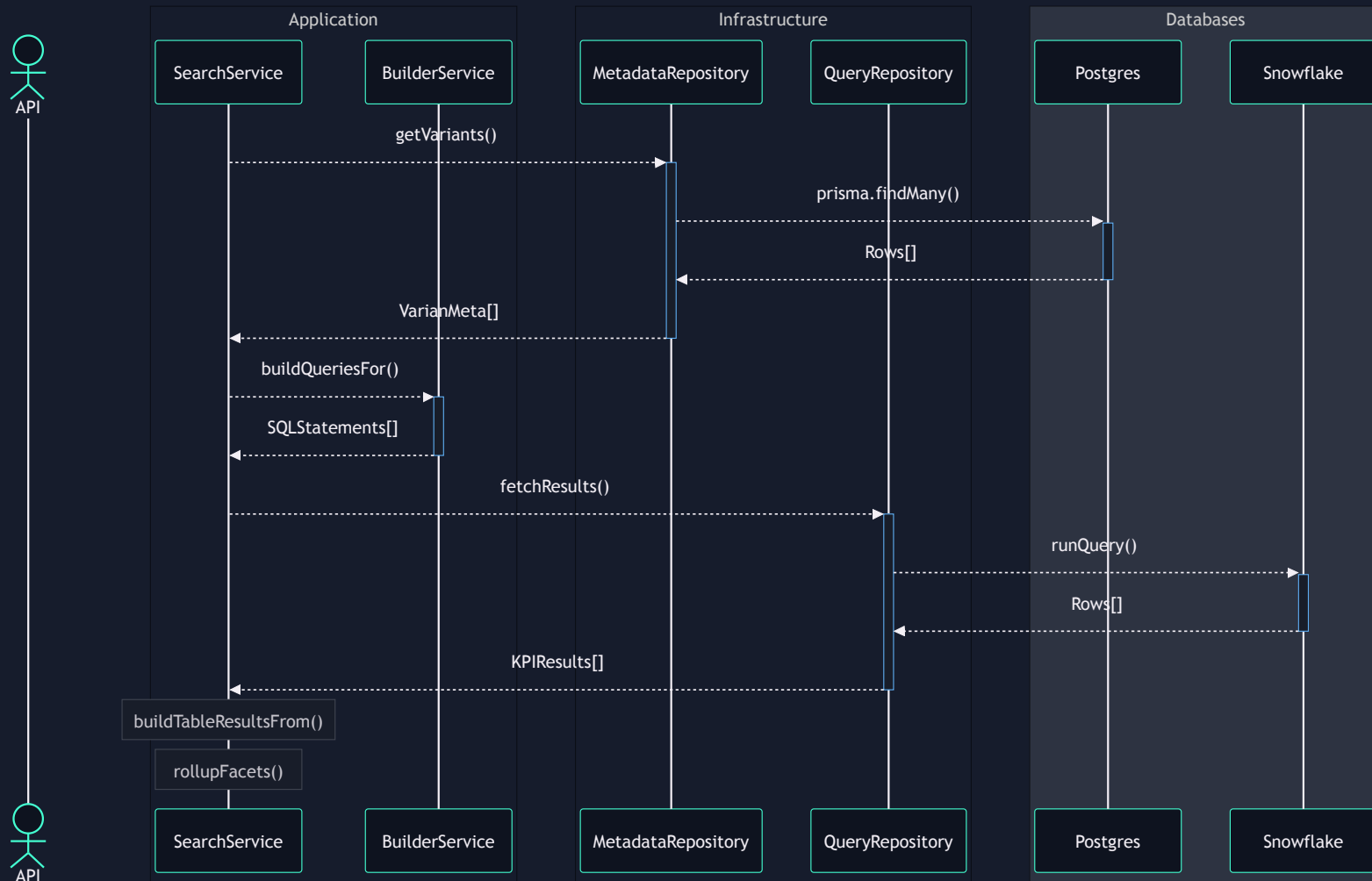
```
 1  import type { ParseError } from "effect/ParseResult";
 2
 3  class DataException extends Data.TaggedError(
 4    "FetchException"
 5  )<{
 6    message: string;
 7    reason?: unknown;
 8  }> {}
 9
10  class BadServerResponse extends Data.TaggedError(
11    "BadServerResponse"
12  )<{
13    message: string;
14  }> {}
15
16  type DataFailure =
17    | DataException
18    | BadServerResponse
19    | ParseError;
```

Real World Examples

Journey Metrics Explorer

Application

SearchService   BuilderService

Infrastructure

MetadataRepository   QueryRepository

Databases

Postgres   Snowflake

API

getVariants()

prisma.findMany()

Rows[]

VarianMeta[]

buildQueriesFor()

SQLStatements[]

fetchResults()

runQuery()

Rows[]

KPIResults[]

buildTableResultsFrom()

rollupFacets()
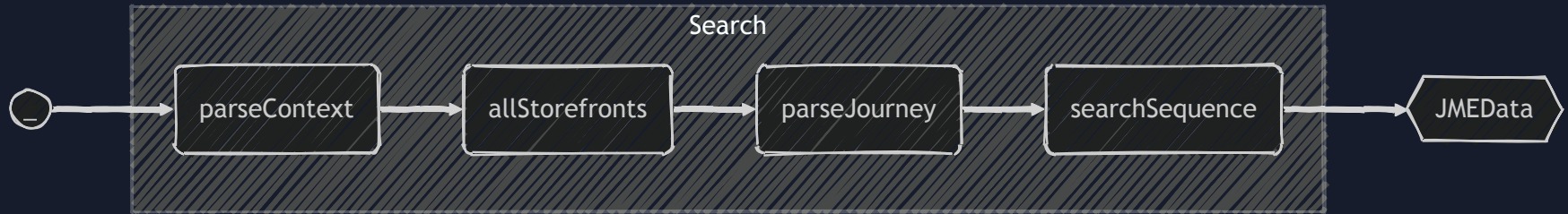
Expectation

~~Expectation~~ Reality
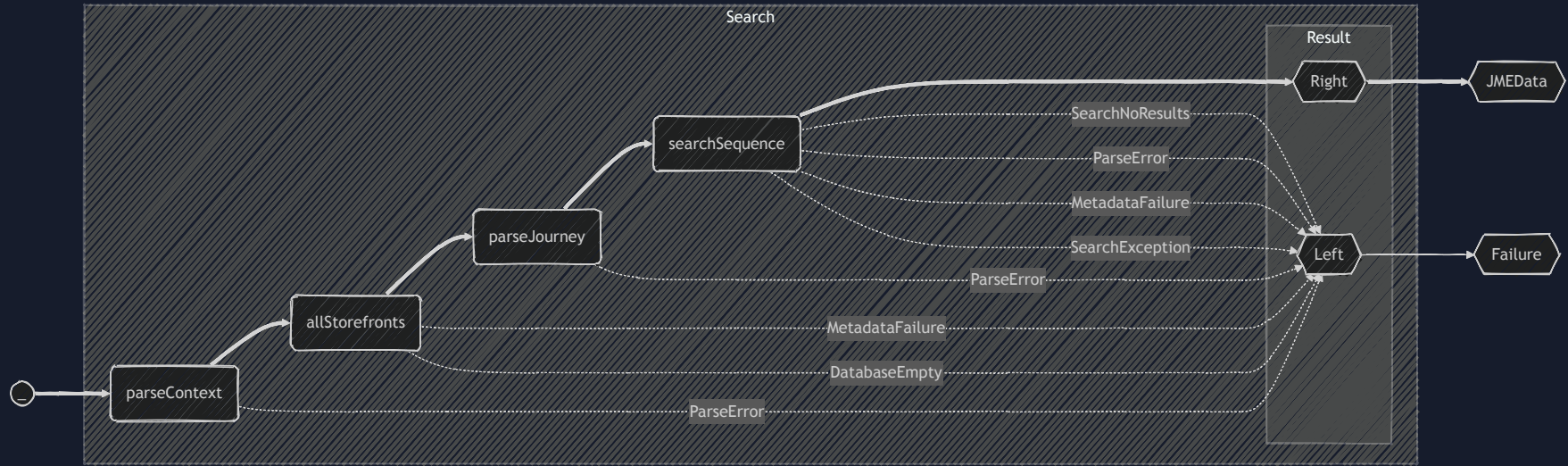
failure	From	description	action
`DatabaseEmpty`	`MetadataRepository` ; `QueryRepository`	No results from the database	try another query
`DatabaseUnavailable`	`MetadataRepository` ; `QueryRepository`	Unable to contact database	try again
`DatabaseValidation`	`MetadataRepository` ; `QueryRepository`	Invalid query to database	contact support with error
`ParseError`	`parseQuery` ; `parseJourney` ; `MetadataRepository` ; `QueryRepository`	Invalid data shape from database or params	contact support with error
`SearchNoResults`	`SearchService`	No results from the search	try another query

Patterns

handleError()

```
1    const handlePrismaError = (error: unknown) ⇒ {
2    if (error && typeof error === "object" && "code" in error) {
3
4      if (error.code === "P2028") {
5        return new DatabaseUnavailable({
6          message: `failed to execute transaction over prisma, `,
7          reason: "meta" in error ? error.meta : error,
8        });
9      }
10
11     return new DatabaseUnavailable({
12       message: `failed to connect to prisma: code ${error.code}`,
13       reason: "meta" in error ? error.meta : error,
14     });
15   }
16   Sentry.captureException(error)
17   return new MetadataException({
18     message: "[RemoteMetadata] unhandled prisma error",
19     reason: error,
20   });
21 };
```

handleError()

```
1   const makeRemoteMetadataRepository = (): MetadataRepository ⇒ ({
2     getVariants: async (args) ⇒ {
3       try {
4         const results =
5           await prisma.metadata.findMany({
6             // ...
7           });
8         if (results.length == 0) {
9           return Either.left(
10            new DatabaseEmpty({
11              message: `no variants found for step_id ${step_id}`,
12            })
13          );
14        }
15        return Schema.decodeUnknownEither(VariantMeta.Array)(results);
16      } catch (err) {
17        return Either.left(
18          handlePrismaError(err)
19        );
20      }
21    },
22  });
```

Data.taggedError

```
1   class DatabaseUnavailable extends Data.TaggedError('DatabaseUnavailable')<{
2     message: string;
3     reason?: unknown;
4   }> {}
5
6   class DatabaseEmpty extends Data.TaggedError('DatabaseEmpty')<{
7     message: string;
8   }> {}
9
10  class MetadataException extends Data.TaggedError("MetadataException")<{
11    message: string,
12    reason?: unknown,
13  }> {}
14
15  class MetadataUninitialized extends Data.TaggedError("MetadataUninitialized")<{
16    message: string,
17  }> {}
18
19  type MetadataFailure = MetadataException | DatabaseUnavailable | DatabaseEmpty;
```

Expand Utility

```typescript
import { Data, Either } from "effect";
import type { ParseError } from "effect/ParseResult";

class DatabaseUnavailable extends Data.TaggedError('DatabaseUnavailable'){}
class DatabaseEmpty extends Data.TaggedError('DatabaseEmpty'){}
class MetadataException extends Data.TaggedError("MetadataException"){}
class ImportException extends Data.TaggedError("ImportException"){}

type MetadataFailure = MetadataException | DatabaseUnavailable | DatabaseEmpty;

// ────────────────────────────────────────────

type Expand<T> = T extends infer U ? U : never;

type _ImportFailure = ImportException | ParseError | MetadataFailure;

type ImportFailure = Expand<ImportException | ParseError | MetadataFailure>;

// ────────────────────────────────────────────

type DataResult = Either.Either<Object, ImportFailure>
```

Resources

- Side-Effects Are The Complexity Iceberg • Kris Jenkins • YOW! 2024

- The most important function in my codebase - Theo - t3.gg

- Effect fro Domains at Vercel | Dillon Murloy (Effect Days 2025)

- Github: neverthrow

- Gist: Theo's preferred way of handling try/catch in Typescript"

- Effect Documentation - Either

Thank you!

Materials · GitHub

Powered by Slidev