





# Result Types

Safe and Sound Error Handling

# Introduction

# "Happy Path Blindness"

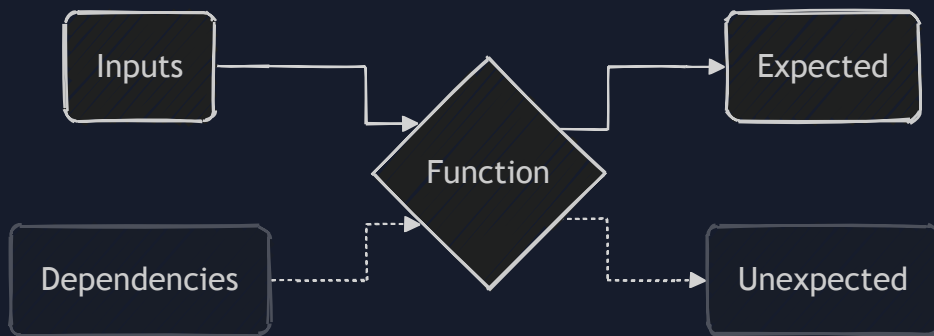
What could go wrong?

-  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

```
1 func divide(a, b int) (int, error) {
2     if b == 0 {
3         return 0, errors.New("division by zero")
4     }
5     return a / b, nil
6 }
7
8 func main() {
9     result, err := divide(10, 0)
10    if err != nil {
11        fmt.Println("Error:", err)
12    } else {
13        fmt.Println("Result:", result)
14    }
15 }
```

# Haskell

```
1  divide :: Int → Int → Either String Int
2  divide a b = if b == 0
3      then Left "division by zero"
4      else Right (a `div` b)
5
6  main :: IO ()
7  main = do
8      let result = divide 10 0
9      case result of
10         Left err → putStrLn $ "Error: " ++ err
11         Right val → putStrLn $ "Result: " ++ show val
```

# Rust

```
1 fn divide(a: i32, b: i32) → Result<i32, String> {
2     if b == 0 {
3         Err("division by zero".to_string())
4     } else {
5         Ok(a / b)
6     }
7 }
8
9 fn main() {
10     match divide(10, 0) {
11         Ok(result) => println!("Result: {}", result),
12         Err(e) => println!("Error: {}", e),
13     }
14 }
```



What is a Result Type?

# tryCatch Block

```
1  const getData = async <T>(from: string) => {
2    try {
3      const url = `/api/data?from=${from}`;
4      const res = await fetch(url);
5      if (!res.ok) {
6        throw new Error(res.statusText);
7      }
8      return (await res.json()) as T;
9    } catch (error) {
10     console.error(error);
11     return [] as T;
12   }
13 };
14
15 const main = async () => {
16   const data = await getData<DataT[]>("2024");
17 };
```

# Discriminate Union

```
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 result = await tryCatch(
12     getData<DataT[]>("2024")
13   );
14   if (result.error) {
15     console.log("Unable to get data");
16     return;
17   }
18   const data = result.data;
19 };
```

```
1  type Success<T> = {
2    data: T;
3    error: null;
4  };
5
6  type Failure<E> = {
7    data: null;
8    error: E;
9  };
10
11 type Result<T, E = Error> =
12   | Success<T>
13   | Failure<E>;
14
15 const tryCatch = async <T, E = Error>(
16   promise: Promise<T>
17 ): Promise<Result<T, E>> => {
18   try {
19     const data = await promise;
20     return { data, error: null };
21   } catch (error) {
22     return { data: null, error: error as E };
23   }
24 };
```

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

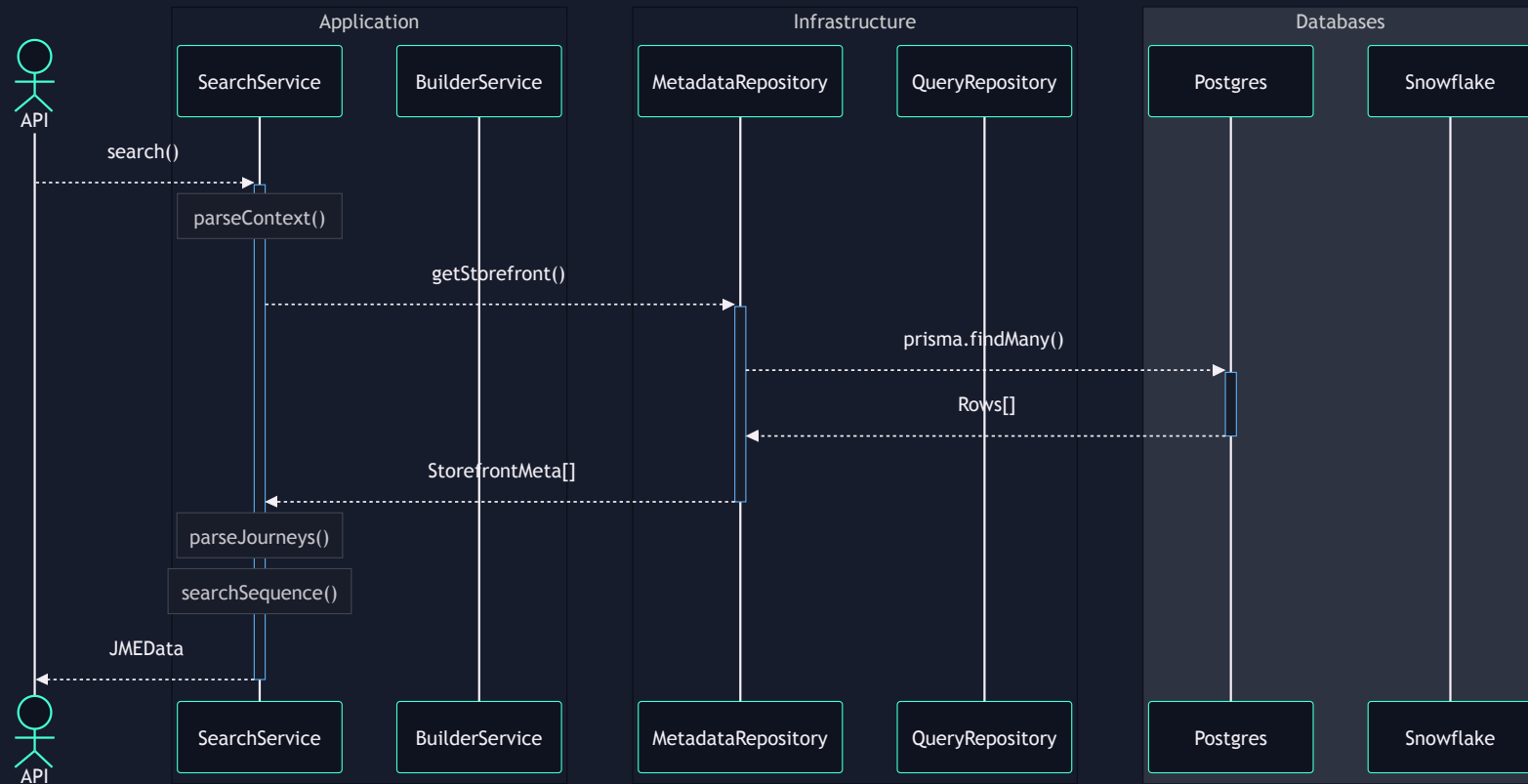
# effect

```
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  export class DataException extends Data.TaggedError(  
4    "FetchException"  
5  )<{  
6    message: string;  
7    reason?: unknown;  
8  }> {}  
9  
10 export class BadServerResponse extends Data.TaggedError(  
11   "BadServerResponse"  
12 )<{  
13   message: string;  
14 }> {}  
15  
16 export type DataFailure =  
17   | DataException  
18   | BadServerResponse  
19   | ParseError;
```

# Real World Examples

Journey Metrics Explorer



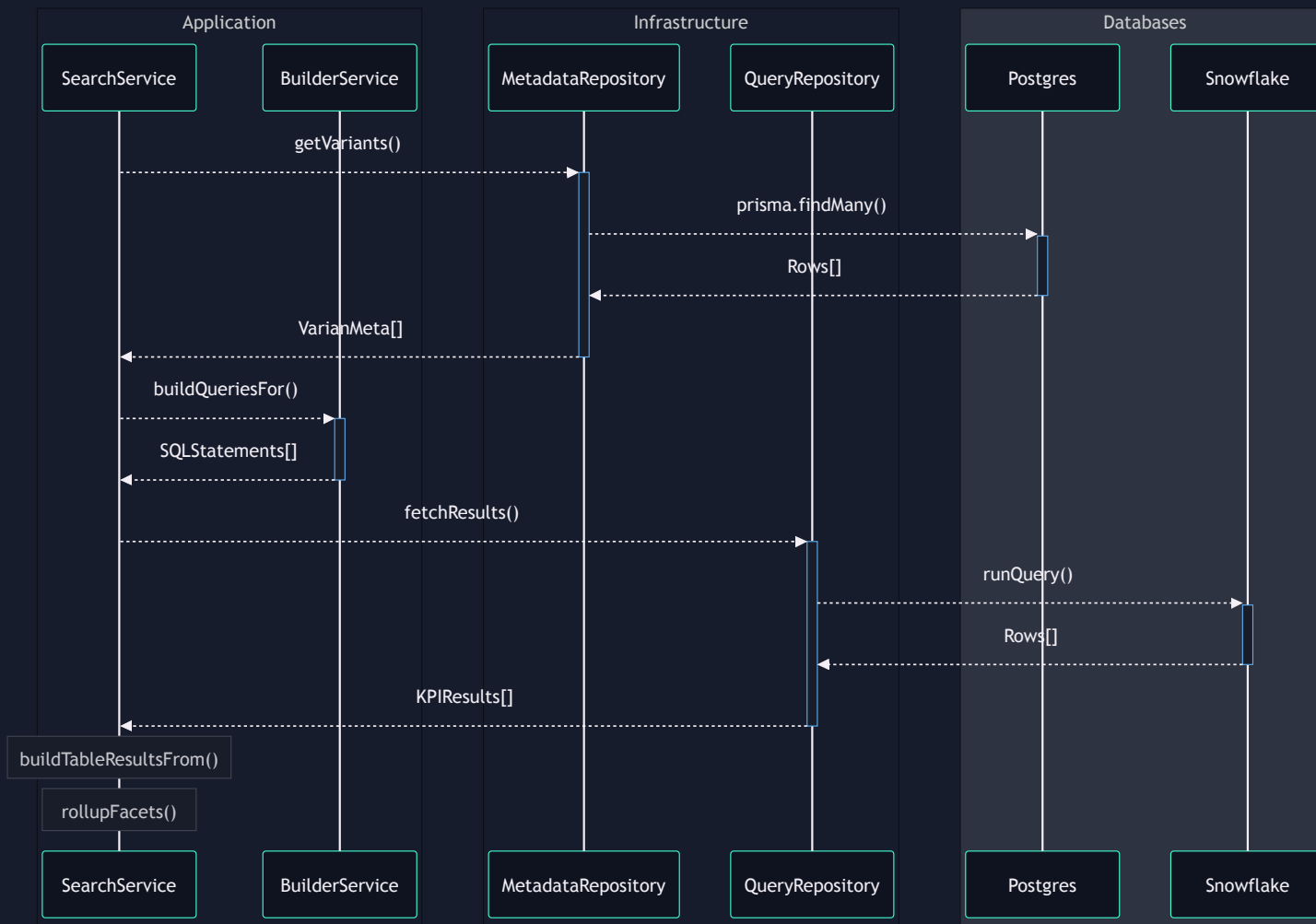




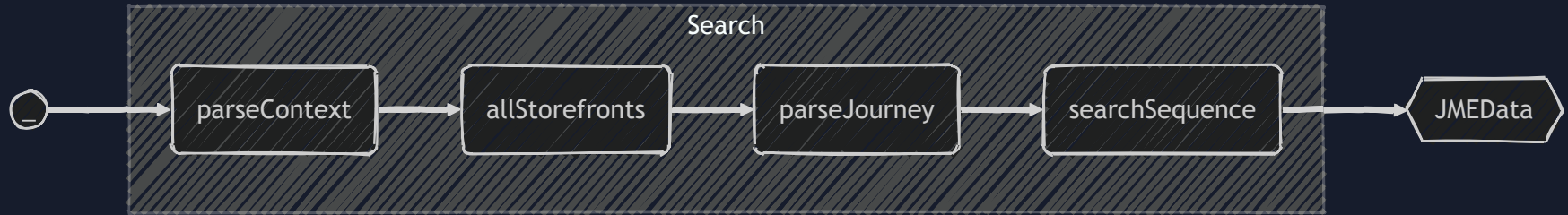
API



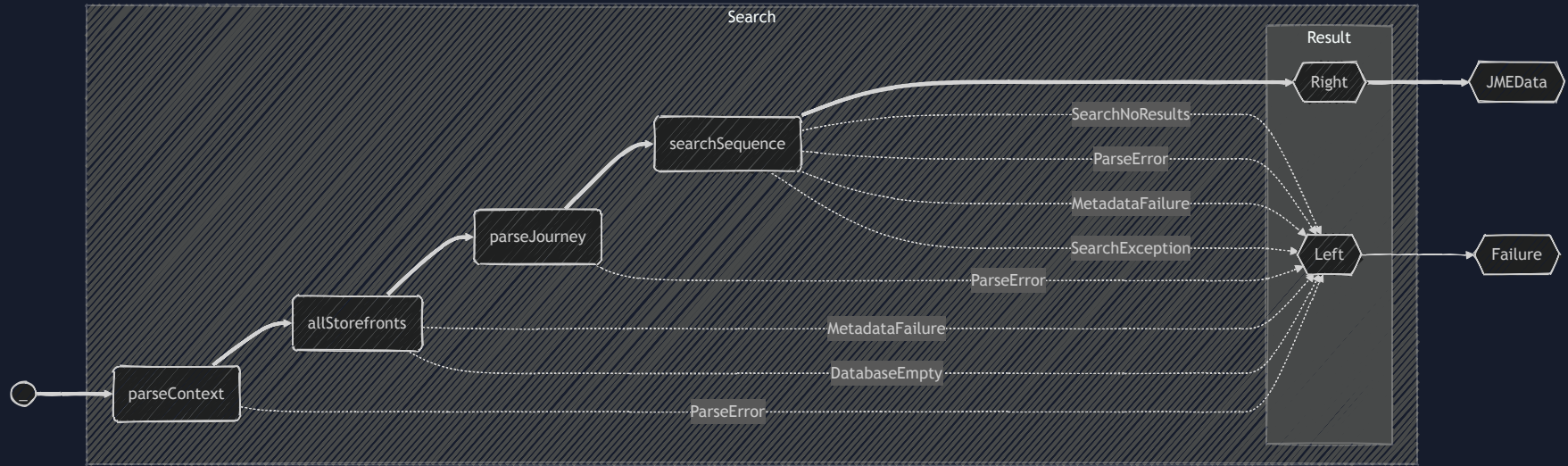
API



# Expectation



# Expectation Reality



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


```
1  import { Data, Either } from "effect";
2  import type { ParseError } from "effect/ParseResult";
3
4  class DatabaseUnavailable extends Data.TaggedError('DatabaseUnavailable'){}
5  class DatabaseEmpty extends Data.TaggedError('DatabaseEmpty'){}
6  class MetadataException extends Data.TaggedError("MetadataException"){}
7  class ImportException extends Data.TaggedError("ImportException"){}
8
9  type MetadataFailure = MetadataException | DatabaseUnavailable | DatabaseEmpty;
10
11  // -----
12
13  type Expand<T> = T extends infer U ? U : never;
14
15  type _ImportFailure = ImportException | ParseError | MetadataFailure;
16
17  type ImportFailure = Expand<ImportException | ParseError | MetadataFailure>;
18
19  // -----
20
21  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