Result Types

Safe and Sound Error Handling

Introduction

The `Result` type helps handle success and error cases without relying on `try/catch`. It makes the data flow more predictable especially in async code, and encourages thinking about both happy and unhappy paths up front.

Turning Typescript into a production-ready language is a long journey. The `Result` type is a step towards that goal, making it easier to write robust and maintainable code.

"Happy Path Blindness"

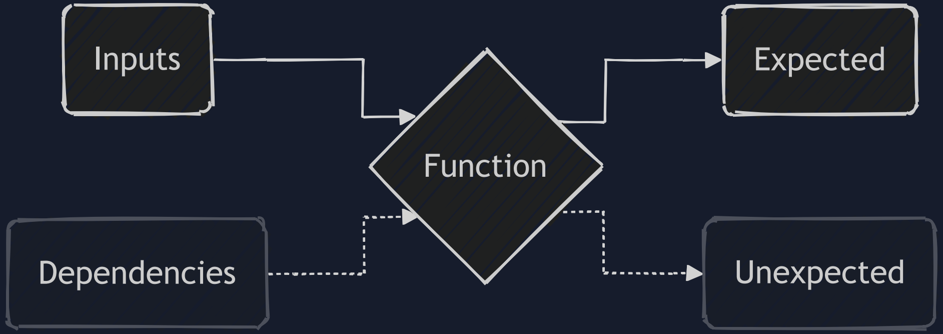
What could go wrong?

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

```
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 data = await getData<DataT[]>("2024");
};
```

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

~ Kris Jenkins



How do other languages do it?

Let’s not reinvent the wheel.

```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?

Something that is either a success or an error, but not both at the same time.

tryCatch Block

```ts
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

```ts
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;
};
```

```ts
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

```typescript
const getData = async <T, E = Error>(
  from: string
): Promise<Either<E, T>> ⇒ {
  try {
    const url = `/api/data?from=${from}`;
    const res = await fetch(url);
    if (!res.ok) {
      return left(new Error(res.statusText) as E);
    }
    const data = (await res.json()) as T;
    return right(data);
  } catch (error) {
    return left(
      new Error("Unable to fetch data") as E
    );
  }
};
```

```typescript
type Left<T> = {
  _tag: "Left";
  left: T;
};

type Right<T> = {
  _tag: "Right";
  right: T;
};

type Either<L, R> = Left<L> | Right<R>;

const left = <L, R>(left: L): Either<L, R> ⇒ {
  return {
    _tag: "Left",
    left,
  };
};
const right = <L, R>(right: R): Either<L, R> ⇒ {
  return {
    _tag: "Right",
    right,
  };
};
```

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

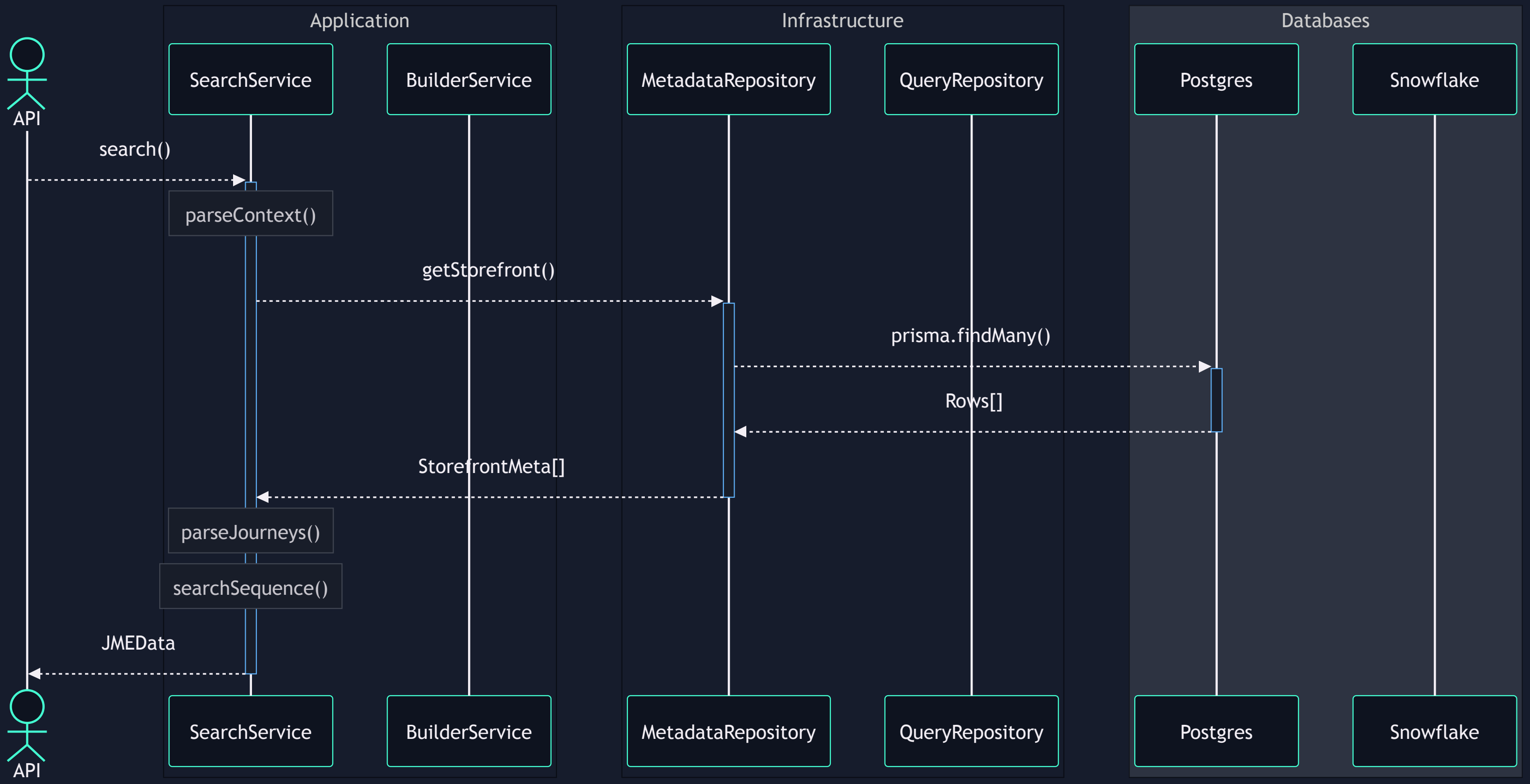
```
const getData = async (
  from: string
): Promise<
  Either.Either<
    ReadonlyArray<typeof Datum.Type>,
    DataFailure
  >
> => {
  try {
    const url = `/api/data?from=${from}`;
    const res = await fetch(url);
    if (!res.ok) {
      return Either.left(
        new BadServerResponse({
          message: `Bad server response: ${res.statusText}`,
        })
      );
    }
    const data = await res.json();
    const parseResult =
      Schema.decodeUnknownEither(
        Schema.Array(Datum)
```

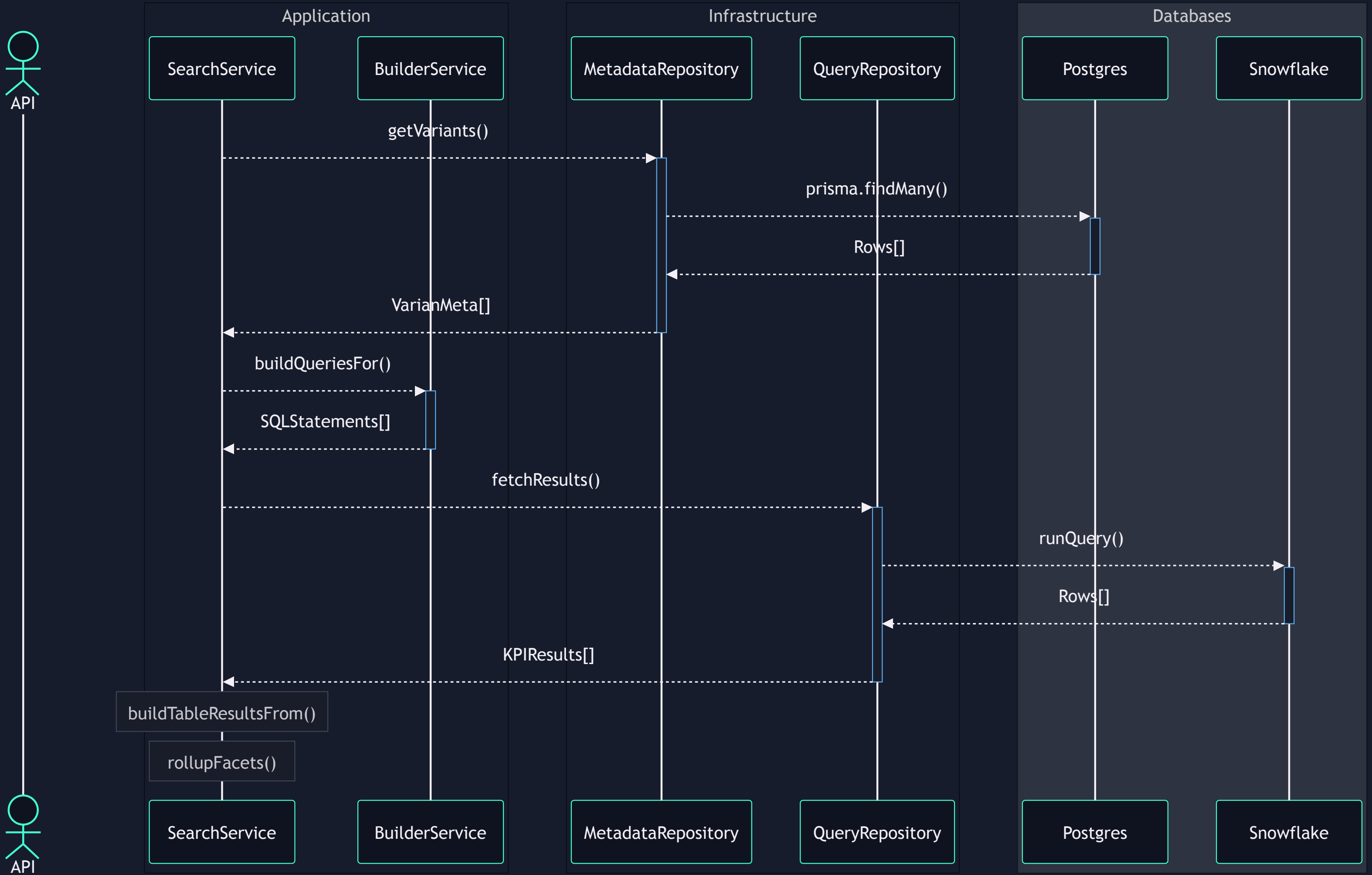
```
import type { ParseError } from "effect/ParseResult'

class DataException extends Data.TaggedError(
  "FetchException"
)<{
  message: string;
  reason?: unknown;
}> {}

class BadServerResponse extends Data.TaggedError(
  "BadServerResponse"
)<{
  message: string;
}> {}

type DataFailure =
  | DataException
  | BadServerResponse
  | ParseError;
```

Real World Examples

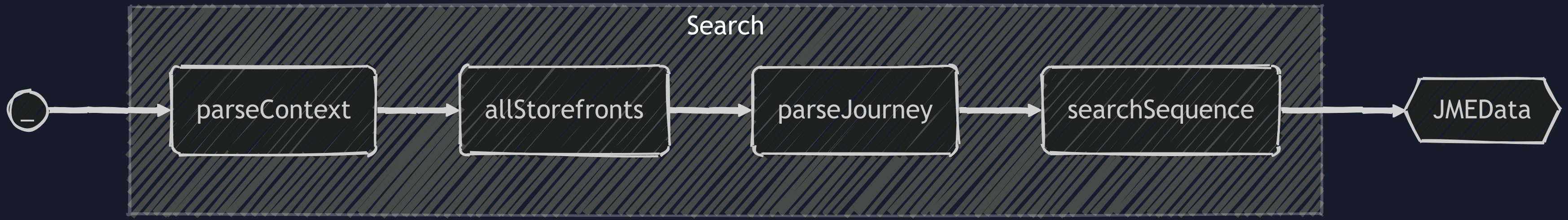
Journey Metrics Explorer

Its "~~just~~" a table.

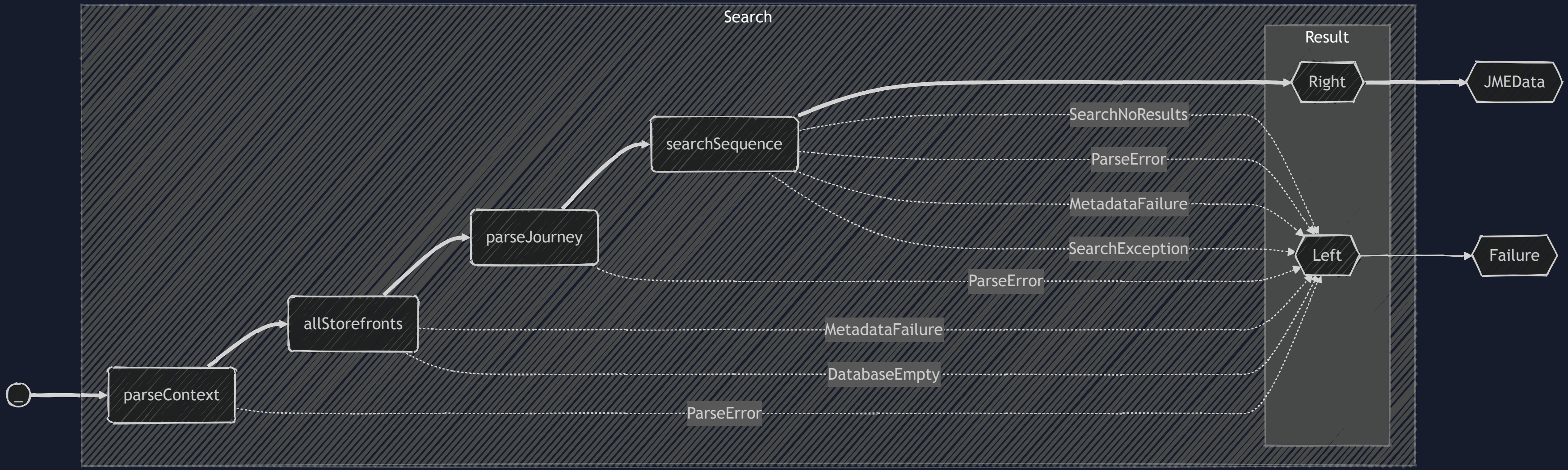

Application
Infrastructure
Databases
API
SearchService
BuilderService
MetadataRepository
QueryRepository
Postgres
Snowflake
search()
parseContext()
getStorefront()
prisma.findMany()
Rows[]
StorefrontMeta[]
parseJourneys()
searchSequence()
JMEData
API
SearchService
BuilderService
MetadataRepository
QueryRepository
Postgres
Snowflake


Application
Infrastructure
Databases
API
SearchService
BuilderService
MetadataRepository
QueryRepository
Postgres
Snowflake
getVariants()
prisma.findMany()
Rows[]
VarianMeta[]
buildQueriesFor()
SQLStatements[]
fetchResults()
runQuery()
Rows[]
KPIResults[]
buildTableResultsFrom()
rollupFacets()
API
SearchService
BuilderService
MetadataRepository
QueryRepository
Postgres
Snowflake

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

What can we do to make our life easier?

handleError()

```
 const handlePrismaError = (error: unknown) => {
 if (error && typeof error == "object" && "code" in error) {

  if (error.code === "P2028") {
   return new DatabaseUnavailable({
    message: `failed to execute transaction over prisma, `,
    reason: "meta" in error ? error.meta : error,
   });
  }

  return new DatabaseUnavailable({
   message: `failed to connect to prisma: code ${error.code}`,
   reason: "meta" in error ? error.meta : error,
  });
 }
 Sentry.captureException(error)
 return new MetadataException({
  message: "[RemoteMetadata] unhandled prisma error",
  reason: error,
 });
};
```

handleError()

```
const makeRemoteMetadataRepository = (): MetadataRepository => ({
  getVariants: async (args) => {
   try {
    const results =
     await prisma.metadata.findMany({
      // ...
     });
    if (results.length == 0) {
     return Either.left(
      new DatabaseEmpty({
       message: `no variants found for step_id ${step_id}`,
      })
     );
    }
    return Schema.decodeUnknownEither(VariantMeta.Array)(results);
   } catch (err) {
    return Either.left(
     handlePrismaError(err)
    );
   }
  },
 });
```

Data.taggedError

```typescript
class DatabaseUnavailable extends Data.TaggedError('DatabaseUnavailable')<{
  message: string;
  reason?: unknown;
}> {}

class DatabaseEmpty extends Data.TaggedError('DatabaseEmpty')<{
  message: string;
}> {}

class MetadataException extends Data.TaggedError("MetadataException")<{
  message: string,
  reason?: unknown,
}> {}

class MetadataUninitialized extends Data.TaggedError("MetadataUninitialized")<{
  message: string,
}> {}

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