Coming From ZIO

Location: 700-other/700-coming-from-zio

Key differences between Effect and ZIO explained, covering the representation of the environment, removal of `Has`, and the absence of predefined type aliases like `UIO`, `URIO`, `RIO`, `Task`, or `IO` in Effect.

If you are coming to Effect from ZIO, there are a few differences to be aware of.

Environment

In Effect, we represent the environment required to run an `Effect` workflow as a union of services:

```
import { Effect } from "effect"

//                                                    v---------v---- `R` is a union of Console | Logger
type Http = Effect.Effect<Response, IOError | HttpError, Console | Logger>

type Response = Record<string, string>

interface IOError {
  readonly _tag: "IOError"
}

interface HttpError {
  readonly _tag: "HttpError"
}

interface Console {
  readonly log: (msg: string) => void
}

interface Logger {
  readonly log: (msg: string) => void
}
```

This may be confusing to folks coming from ZIO, where the environment is represented as an intersection of services:

```
type Http = ZIO[Console with Logger, IOError, Response]
```

Rationale

The rationale for using a union to represent the environment required by an `Effect` workflow boils down to our desire to remove `Has` as a wrapper for services in the environment (similar to what was achieved in ZIO 2.0).

To be able to remove `Has` from Effect, we had to think a bit more structurally given that TypeScript is a structural type system. In TypeScript, if you have a type `A & B` where there is a structural conflict between `A` and `B`, the type `A & B` will reduce to `never`.

```
// @errors: 2322
export interface A {
  readonly prop: string
}

export interface B {
  readonly prop: number
}

const ab: A & B = {
  prop: ""
}
```

In previous versions of Effect, intersections were used for representing an environment with multiple services. The problem with using intersections (i.e. `A & B`) is that there could be multiple services in the environment that have functions and properties named in the same way. To remedy this, we wrapped services in the `Has` type (similar to ZIO 1.0), so you would have `Has<A> & Has<B>` in your environment.

In ZIO 2.0, the *contravariant* `R` type parameter of the `ZIO` type (representing the environment) became fully phantom, thus allowing for removal of the `Has` type. This significantly improved the clarity of type signatures as well as removing another "stumbling block" for new users.

To facilitate removal of `Has` in Effect, we had to consider how types in the environment compose. By the rule of composition, contravariant parameters composed as an intersection (i.e. with `&`) are equivalent to covariant parameters composed together as a union (i.e. with `|`) for purposes of assignability. Based upon this fact, we decided to diverge from ZIO and make the `R` type parameter *covariant* given `A | B` does not reduce to `never` if `A` and `B` have conflicts.

From our example above:

```
export interface A {
  readonly prop: string
}

export interface B {
  readonly prop: number
```

```
}

const ab: A | B = {
  prop: ""
}
```

Representing `R` as a covariant type parameter containing the union of services required by an `Effect` workflow allowed us to remove the requirement for `Has`.

Type Aliases

In Effect, there are no predefined type aliases such as `UIO`, `URIO`, `RIO`, `Task`, or `IO` like in ZIO.

The reason for this is that type aliases are lost as soon as you compose them, which renders them somewhat useless unless you maintain **multiple** signatures for **every** function. In Effect, we have chosen not to go down this path. Instead, we utilize the `never` type to indicate unused types.

It's worth mentioning that the perception of type aliases being quicker to understand is often just an illusion. In Effect, the explicit notation `Effect<A>` clearly communicates that only type `A` is being used. On the other hand, when using a type alias like `RIO<R, A>`, questions arise about the type `E`. Is it `unknown`? `never`? Remembering such details becomes challenging.

FAQ

Location: 700-other/100-faq

Explore common questions about Effect, including type extraction, sync/async behavior, comparison with fp-ts, understanding flatMap, and the absence of ZIO-like type aliases. Learn about configuring layers and the flexibility of accepting arguments to influence service construction in layers.

Effect

Q: Is it possible to extract the types from an Effect?

A: By using the utility types `Effect.Effect.Context`, `Effect.Effect.Error`, and `Effect.Effect.Success`, we can extract the corresponding types from the `program` Effect. In this example, we extract the context type (`R`), the error type (`E`), and the success type (`A`).

```
import { Effect, Context } from "effect"

class Random extends Context.Tag("Random")<
  Random,
  {
    readonly next: Effect.Effect<number>
  }
>() {}

declare const program: Effect.Effect<number, string, Random>

// type R = Random
type R = Effect.Effect.Context<typeof program>

// type E = string
type E = Effect.Effect.Error<typeof program>

// type A = number
type A = Effect.Effect.Success<typeof program>
```

Q: Is there a way to determine whether an Effect is synchronous or asynchronous in advance?

A: No, there isn't a straightforward way to statically determine if an Effect is synchronous or asynchronous. We did explore this idea in earlier versions of Effect, but we decided against it for a few important reasons:

1. **Complexity:** Introducing this feature to track sync/async behavior in the type system would make Effect more complex to use and limit its composability.

2. **Safety Concerns:** We experimented with different approaches to track asynchronous Effects, but they all resulted in a worse developer experience without significantly improving safety. Even with fully synchronous types, we needed to support a `fromCallback` combinator to work with APIs using Continuation-Passing Style (CPS). However, at the type level, it's impossible to guarantee that such a function is always called immediately and not deferred.

In practice, it's important to note that you should typically only run Effects at the edges of your application. If your application is entirely based on Effect, it usually involves a single call to the `main` effect. In such cases, the best approach is to use `runPromise` or `runFork` for most executions. Synchronous execution is a special case and should be considered an edge case, used only when asynchronous execution is not possible. So, our recommendation is to use `runPromise` or `runFork` whenever you can and resort to `runSync` only when absolutely necessary.

Q: I'm familiar with `flatMap` in JavaScript/TypeScript from its usage on the `Array` prototype. Why do I see it in modules provided by Effect?

A: Many JavaScript / TypeScript engineers are familiar with the term `flatMap` due to it's presence as a method on the `Array` prototype. Therefore it may be confusing to see `flatMap` methods exported from many of the modules that Effect provides.

The `flatMap` operation can actually be used to describe a more generic data transformation. Let's consider for a moment a generic data type that we will call `F`. `F` will also be a container for elements, so we can further refine our representation of `F` to `F<A>` which states that `F` holds some information about some data of type `A`.

If we have an `F<A>` and we want to transform to an `F<B>`, we could use the `map` operation:

```
map: <A, B>(fa: F<A>, f: (a: A) => B) => F<B>
```

But what if we have some function that returns an `F<B>` instead of just `B`? We can't use `map` because we would end up with a `F<F<B>>` instead of an `F<B>`. What we really want is some operator that allows us to `map` the data and then `flatten` the result.

This exact situation describes a `flatMap` operation:

```
flatMap: <A, B>(fa: F<A>, f: (a: A) => F<B>) => F<B>
```

You can also see how this directly applies to `Array`'s `flatMap` if we replace our generic data type `F` with the concrete data type of `Array`:

```
flatMap: <A, B>(fa: Array<A>, f: (a: A) => Array<B>) => Array<B>
```

Q: I can't seem to find any type aliases for Effect. Do any exist? I'm looking for something similar to ZIO's `UIO` / `URIO` / `RIO` / `Task` / `IO`. If not, have you considered adding them?

A: In Effect, there are no predefined type aliases such as `UIO`, `URIO`, `RIO`, `Task`, or `IO` like in ZIO.

The reason for this is that type aliases are lost as soon as you compose them, which renders them somewhat useless unless you maintain **multiple** signatures for **every** function. In Effect, we have chosen not to go down this path. Instead, we utilize the `never` type to indicate unused types.

It's worth mentioning that the perception of type aliases being quicker to understand is often just an illusion. In Effect, the explicit notation `Effect<A>` clearly communicates that only type `A` is being used. On the other hand, when using a type alias like `RIO<R, A>`, questions arise about the type `E`. Is it `unknown`? `never`? Remembering such details becomes challenging.

Comparison with fp-ts

Q: What are the main differences between Effect and fp-ts?

A: The main differences between Effect and fp-ts are:

- Effect offers more flexible and powerful dependency management.
- Effect provides built-in services like `Clock`, `Random`, and `Tracer`, which fp-ts lacks.
- Effect includes dedicated testing tools like `TestClock`, while fp-ts does not offer specific testing utilities.
- Effect supports interruptions for canceling computations, whereas fp-ts does not have built-in interruption support.
- Effect has built-in tools to handle defects and unexpected failures, while fp-ts lacks specific defect management features.
- Effect leverages fiber-based concurrency for efficient and lightweight concurrent computations, which fp-ts does not provide.
- Effect includes built-in support for customizable retrying of computations, while fp-ts does not have this feature.
- Effect offers built-in logging, scheduling, caching, and more, which fp-ts does not provide.

For a more detailed comparison, you can refer to the Effect vs fp-ts documentation.

Bundle Size Comparison Between Effect and fp-ts

Q: I compared the bundle sizes of two simple programs using Effect and fp-ts. Why does Effect have a larger bundle size?

A: It's natural to observe different bundle sizes because Effect and fp-ts are distinct systems designed for different purposes. Effect's bundle size is larger due to its included fiber runtime, which is crucial for its functionality. While the initial bundle size may seem large, the overhead amortizes as you use Effect.

Q: Should I be concerned about the bundle size difference when choosing between Effect and fp-ts?

A: Not necessarily. Consider the specific requirements and benefits of each library for your project.

Glossary

Location: 700-other/200-glossary

Explore key concepts in Effect, such as context, dual APIs, distributed workflows, expected errors, fibers, interruption, local workflows, effect pipelines, schedules, services, tags, and unexpected errors. Understand their significance in managing dependencies, controlling concurrency, and handling errors within effectful computations.

Context

In `Effect`, context refers to a container that holds important contextual data required for the execution of effects. It's a conceptual construct that enables effects to access and utilize contextual data within their execution scope.

Think of context as a `Map<Tag, Impl>` that associates `Tag`s with their corresponding implementations. By providing the appropriate context to an effect, the effect gains access to the specific contextual data it needs. This allows the effect to perform operations that rely on that data.

Context plays a vital role in managing dependencies and facilitating the composition of effects in a modular and flexible manner. It ensures that effects have access to the necessary data they depend on, making it easier to organize and maintain code.

Dual (API)

In the context of APIs, "dual" refers to a function that supports both the "data-first" and "data-last" variants. It means that the API can be used interchangeably with either style, depending on the developer's preference.

One example of a dual API is the `andThen` function of the `Effect` data type.

In the "data-first" variant, the effect is passed as the first argument to `andThen`:

```
import { Effect } from "effect"

Effect.andThen(Effect.succeed(1), (n) => n + 1)
```

while in the "data-last" variant, the effect is passed as the first argument to the `pipe` function, followed by the `andThen` function:

```
import { Effect, pipe } from "effect"

pipe(
  Effect.succeed(1),
  Effect.andThen((n) => n + 1)
)

// or

Effect.succeed(1).pipe(Effect.andThen((n) => n + 1))
```

Distributed workflow

Things that may execute across multiple execution boundaries.

Expected Errors

Also referred to as *failures*, *typed errors* or *recoverable errors*.

These are errors that developers anticipate and expect as part of normal program execution. Expected errors are tracked at the type level by the `Effect` data type in the "Error" channel:

```
Effect<Value, Error, Context>
```

Fiber

A "fiber" is a small unit of work or a lightweight thread of execution. It represents a specific computation or an effectful operation in a program. Fibers are used to manage concurrency and asynchronous tasks.

Think of a fiber as a worker that performs a specific job. It can be started, paused, resumed, and even interrupted. Fibers are useful when we want to perform multiple tasks simultaneously or handle long-running operations without blocking the main program.

By using fibers, developers can control and coordinate the execution of tasks, allowing for efficient multitasking and responsiveness in their applications.

To summarize:

- An `Effect` is a higher-level concept that describes an effectful computation. It is lazy and immutable, meaning it represents a computation that may produce a value or fail but does not immediately execute.
- A fiber, on the other hand, represents the running execution of an `Effect`. It can be interrupted or awaited to retrieve its result. Think of it as a way to control and interact with the ongoing computation.

Interruption

Interruption errors occur when the execution of a running fiber is deliberately interrupted.

Local workflow

Things that execute within a single execution boundary.

Pipeline (of Effects)

A "pipeline" refers to a series of sequential operations performed on `Effect` values to achieve a desired result. When working with `Effect`, a pipeline typically consists of operations such as mapping, flat-mapping, filtering, and error handling, among others. Each operation in the pipeline takes an input `Effect` and produces an output `Effect`, which becomes the input for the next operation in the sequence.

Schedule

A Schedule is an immutable value that defines a strategy for repeating or retrying an effectful operation. Schedules can be composed and combined together to create more complex recurrence patterns. This allows for flexible and customizable scheduling strategies.

Service

A Service is an interface that defines a set of operations or functionality. Services encapsulate specific capabilities or behaviors that can be utilized by effects. By providing service implementations, we can enhance the capabilities of effects and enable them to interact with external systems, perform IO operations, or access shared resources. Services are usually associated with Tags, which allow effects to locate and access the corresponding service implementation during runtime.

Tag

In Effect, a Tag is a unique marker that represents a specific value in a `Context`. It is used when you need to uniquely identify something in a "bag of type-safe things". Tags are similar to map keys but come with associated types. In the context of Effect, a Tag is often used to represent a specific service.

Tags serve as keys that allow Effect to locate and use the corresponding service implementation at runtime. They play a crucial role in managing dependencies in a type-safe manner and providing the required services for the smooth execution of your effects.

For examples of how to use Tags in creating a simple service, check out Creating a Simple Service.

Unexpected Errors

Also referred to as *defects*, *untyped errors*, or *unrecoverable errors*.

These are errors that occur unexpectedly and are not part of the intended program flow.

Other

Location: 700-other/index

Other

Either

Location: 700-other/300-data-types/either

Explore the `Either` data type in Effect for representing exclusive values using `Left` and `Right`. Learn to create Eithers, use guards for checking types, and perform pattern matching. Discover operations like mapping over `Right` and `Left` values, and understand interoperability with the `Effect` type. Master the expressive and versatile features of `Either` for error handling and result representation in your applications.

The `Either` data type represents two exclusive possible values: an `Either<R, L>` can be either a `Right` value or a `Left` value, where `R` represents the type of the `Right` value and `L` represents the type of the `Left` value.

Understanding Either and Exit

- Either is primarily used as a **simple discriminated union** and is not recommended as the main result type for operations requiring detailed error information.
- Exit is the preferred **result type** within Effect for capturing comprehensive details about failures. It encapsulates the outcomes of effectful computations, distinguishing between success and various failure modes, such as errors, defects and interruptions.

Creating Eithers

You can create an `Either` using the `Either.right` and `Either.left` constructors.

- The `Either.right` function takes a value of type `R` and constructs a `Either<R, never>`:

```
import { Either } from "effect"

const rightValue = Either.right(42)
```

- The `Either.left` function takes a value of type `L` and constructs a `Either<never, L>`:

```
import { Either } from "effect"

const leftValue = Either.left("not a number")
```

Guards

You can determine whether an `Either` is a `Left` or a `Right` by using the `Either.isLeft` and `Either.isRight` guards:

```
import { Either } from "effect"

const foo = Either.right(42)

if (Either.isLeft(foo)) {
  console.log(`The left value is: ${foo.left}`)
} else {
  console.log(`The Right value is: ${foo.right}`)
}
// Output: "The Right value is: 42"
```

Pattern Matching

The `Either.match` function allows you to handle different cases of an `Either` by providing separate callbacks for each case:

```
import { Either } from "effect"

const foo = Either.right(42)

const message = Either.match(foo, {
  onLeft: (left) => `The left value is: ${left}`,
  onRight: (right) => `The Right value is: ${right}`
})

console.log(message) // Output: "The Right value is: 42"
```

<Idea> Using `match` instead of `isLeft` or `isRight` can be more expressive and provide a clear way to handle both cases of an `Either`. </Idea>

Working with Either

Once you have an `Either`, there are several operations you can perform on it.

Mapping over the Right Value

You can use the `Either.map` function to transform the `Right` value of an `Either`. The `Either.map` function takes a transformation function that maps the `Right` value.

If the `Either` value is a `Left` value, the transformation function is **ignored**, and the `Left` value is returned unchanged.

Example

```
import { Either } from "effect"

Either.map(Either.right(1), (n) => n + 1) // right(2)

Either.map(Either.left("not a number"), (n) => n + 1) // left("not a number")
```

Mapping over the Left Value

You can use the `Either.mapLeft` function to transform the `Left` value of an Either. The `mapLeft` function takes a transformation function that maps the `Left`.

If the `Either` value is a `Right` value, the transformation function is **ignored**, and the `Right` value is returned unchanged.

Example

```
import { Either } from "effect"

Either.mapLeft(Either.right(1), (s) => s + "!") // right(1)

Either.mapLeft(Either.left("not a number"), (s) => s + "!") // left("not a number!")
```

Mapping over Both Values

You can use the `Either.mapBoth` function to transform both the `Left` and `Right` values of an `Either`. The `mapBoth` function takes two transformation functions: one for the `Left` value and one for the `Right` value.

Example

```
import { Either } from "effect"

Either.mapBoth(Either.right(1), {
  onLeft: (s) => s + "!",
  onRight: (n) => n + 1
}) // right(2)
```

```
Either.mapBoth(Either.left("not a number"), {
  onLeft: (s) => s + "!",
  onRight: (n) => n + 1
}) // left("not a number!")
```

Interop with Effect

The `Either` type is a subtype of the `Effect` type, which means that it can be seamlessly used with functions from the `Effect` module. These functions are primarily designed to work with `Effect` values, but they can also handle `Either` values and process them correctly.

In the context of `Effect`, the two members of the `Either` type are treated as follows:

- `Left<L>` is equivalent to `Effect<never, L>`
- `Right<R>` is equivalent to `Effect<R>`

To illustrate this interoperability, let's consider the following example:

```
import { Effect, Either } from "effect"

const head = <A>(array: ReadonlyArray<A>): Either.Either<A, string> =>
  array.length > 0 ? Either.right(array[0]) : Either.left("empty array")

const foo = Effect.runSync(Effect.andThen(Effect.succeed([1, 2, 3]), head))
console.log(foo) // Output: 1

const bar = Effect.runSync(Effect.andThen(Effect.succeed([]), head)) // throws "empty array"
```

Combining Two or More Eithers

The `Either.zipWith` function allows you to combine two `Either` values using a provided function. It creates a new `Either` that holds the combined value of both original `Either` values.

```
import { Either } from "effect"

const maybeName: Either.Either<string, Error> = Either.right("John")
const maybeAge: Either.Either<number, Error> = Either.right(25)

const person = Either.zipWith(maybeName, maybeAge, (name, age) => ({
  name: name.toUpperCase(),
  age
}))

console.log(person)
/*
Output:
{ _id: 'Either', _tag: 'Right', right: { name: 'JOHN', age: 25 } }
*/
```

The `Either.zipWith` function takes three arguments:

- The first `Either` you want to combine
- The second `Either` you want to combine
- A function that takes two arguments, which are the values held by the two `Either`, and returns the combined value

It's important to note that if either of the two `Either` values is `Left`, the resulting `Either` will also be `Left`, containing the value of the first encountered `Left`:

```
import { Either } from "effect"

const maybeName: Either.Either<string, Error> = Either.right("John")
const maybeAge: Either.Either<number, Error> = Either.left(
  new Error("Oh no!")
)

const person = Either.zipWith(maybeName, maybeAge, (name, age) => ({
  name: name.toUpperCase(),
  age
}))

console.log(person)
/*
Output:
{ _id: 'Either', _tag: 'Left', left: new Error("Oh no!") }
*/
```

If you need to combine two or more `Either`s without transforming the values they hold, you can use `Either.all`, which takes a collection of `Either`s and returns an `Either` with the same structure.

- If a tuple is provided, the returned `Either` will contain a tuple with the same length.
- If a struct is provided, the returned `Either` will contain a struct with the same keys.
- If an iterable is provided, the returned `Either` will contain an array.

```
import { Either } from "effect"

const maybeName: Either.Either<string, Error> = Either.right("John")
const maybeAge: Either.Either<number, Error> = Either.right(25)

const tuple = Either.all([maybeName, maybeAge])

const struct = Either.all({ name: maybeName, age: maybeAge })
```

Note that if one or more `Either` is a `Left`, then the first encountered `Left` will be returned:

```
import { Either } from "effect"

const maybeName: Either.Either<string, Error> = Either.left(
  new Error("name not found")
)
const maybeAge: Either.Either<number, Error> = Either.left(
  new Error("age not found")
)

const tuple = Either.all([maybeName, maybeAge])

console.log(tuple)
/*
Output:
{ _id: 'Either', _tag: 'Left', left: new Error("name not found") }
*/
```

gen

Similar to Effect.gen, there's also `Either.gen`, which provides a convenient syntax, akin to async/await, for writing code involving `Either` and using generators.

Let's revisit the previous example, this time using `Either.gen` instead of `Either.zipWith`:

```
import { Either } from "effect"

const maybeName: Either.Either<string, Error> = Either.right("John")
const maybeAge: Either.Either<number, Error> = Either.right(25)

const person = Either.gen(function* () {
  const name = (yield* maybeName).toUpperCase()
  const age = yield* maybeAge
  return { name, age }
})

console.log(person)
/*
Output:
{ _id: 'Either', _tag: 'Right', right: { name: 'JOHN', age: 25 } }
*/
```

Once again, if either of the two `Either` values is `Left`, the resulting `Either` will also be `Left`:

```
import { Either } from "effect"

const maybeName: Either.Either<string, Error> = Either.right("John")
const maybeAge: Either.Either<number, Error> = Either.left(
  new Error("Oh no!")
)

const person = Either.gen(function* () {
  const name = (yield* maybeName).toUpperCase()
  const age = yield* maybeAge
  return { name, age }
})

console.log(person)
/*
Output:
{ _id: 'Either', _tag: 'Left', left: new Error("Oh no!") }
*/
```

Option

Location: 700-other/300-data-types/option

Master the versatile `Option` data type for handling optional values. Learn to create, model optional properties, and utilize guards. Discover powerful functions like `Option.map`, `Option.flatMap`, and explore seamless interop with nullable types and the Effect module. Also, delve into fallback strategies, working with nullable types, combining options, and much more.

The `Option` data type is used to represent optional values. An `Option` can be either `Some`, which contains a value, or `None`, which indicates the absence of a value.

The `Option` type is versatile and can be applied in various scenarios, including:

- Using it for initial values
- Returning values from functions that are not defined for all possible inputs (referred to as "partial functions")
- Managing optional fields in data structures
- Handling optional function arguments

Creating Options

some

The `Option.some` constructor takes a value of type `A` and returns an `Option<A>` that holds that value:

```
import { Option } from "effect"

const value = Option.some(1) // An Option holding the number 1
```

none

On the other hand, the `Option.none` constructor returns an `Option<never>`, representing the absence of a value:

```
import { Option } from "effect"

const noValue = Option.none() // An Option holding no value
```

liftPredicate

Sometimes you need to create an `Option` based on a predicate, such as checking if a value is positive.

Here's how you can do this explicitly using `Option.none` and `Option.some`

```
import { Option } from "effect"

const isPositive = (n: number) => n > 0

const parsePositive = (n: number): Option.Option<number> =>
  isPositive(n) ? Option.some(n) : Option.none()
```

The same result can be achieved more concisely using `Option.liftPredicate`

```
import { Option } from "effect"

const isPositive = (n: number) => n > 0

const parsePositive = Option.liftPredicate(isPositive)
```

Modeling Optional Properties

Let's look at an example of a `User` model where the `"email"` property is optional and can have a value of type `string`. To represent this, we can use the `Option<string>` type:

```
import { Option } from "effect"

interface User {
  readonly id: number
  readonly username: string
  readonly email: Option.Option<string>
}
```

<Warning> Optionality only applies to the value of the property. The key `"email"` will always be present in the object, regardless of whether it has a value or not. </Warning>

Now, let's see how we can create instances of `User` with and without an email:

```
import { Option } from "effect"

interface User {
  readonly id: number
  readonly username: string
  readonly email: Option.Option<string>
}

// ---cut---
const withEmail: User = {
  id: 1,
  username: "john_doe",
  email: Option.some("john.doe@example.com")
}

const withoutEmail: User = {
  id: 2,
```

```
  username: "jane_doe",
  email: Option.none()
}
```

Guards

You can determine whether an `Option` is a `Some` or a `None` by using the `isSome` and `isNone` guards:

```
import { Option } from "effect"

const foo = Option.some(1)

console.log(Option.isSome(foo)) // Output: true

if (Option.isNone(foo)) {
  console.log("Option is empty")
} else {
  console.log(`Option has a value: ${foo.value}`)
}
// Output: "Option has a value: 1"
```

Matching

The `Option.match` function allows you to handle different cases of an `Option` value by providing separate actions for each case:

```
import { Option } from "effect"

const foo = Option.some(1)

const result = Option.match(foo, {
  onNone: () => "Option is empty",
  onSome: (value) => `Option has a value: ${value}`
})

console.log(result) // Output: "Option has a value: 1"
```

<Idea> Using `match` instead of `isSome` or `isNone` can be more expressive and provide a clear way to handle both cases of an `Option`. </Idea>

Working with Option

map

The `Option.map` function allows you to transform the value inside an `Option` without having to unwrap and wrap the underlying value. Let's see an example:

```
import { Option } from "effect"

const maybeIncremented = Option.map(Option.some(1), (n) => n + 1) // some(2)
```

The convenient aspect of using `Option` is how it handles the absence of a value, represented by `None`:

```
import { Option } from "effect"

const maybeIncremented = Option.map(Option.none(), (n) => n + 1) // none()
```

Despite having `None` as the input, we can still operate on the `Option` without encountering errors. The mapping function `(n) => n + 1` is not executed when the `Option` is `None`, and the result remains `none` representing the absence of a value.

flatMap

The `Option.flatMap` function works similarly to `Option.map`, but with an additional feature. It allows us to sequence computations that depend on the absence or presence of a value in an `Option`.

Let's explore an example that involves a **nested** optional property. We have a `User` model with an optional `address` field of type `Option<Address>`:

```
interface Address {
  readonly city: string
  readonly street: Option.Option<string>
}

// ---cut---
import { Option } from "effect"

interface User {
  readonly id: number
  readonly username: string
  readonly email: Option.Option<string>
  readonly address: Option.Option<Address>
}
```

The `address` field itself contains a nested optional property called `street` of type `Option<string>`:

```
import { Option } from "effect"

// ---cut---
interface Address {
  readonly city: string
  readonly street: Option.Option<string>
}
```

We can use `Option.flatMap` to extract the `street` property from the `address` field.

```
import { Option } from "effect"

interface Address {
  readonly city: string
  readonly street: Option.Option<string>
}

interface User {
  readonly id: number
  readonly username: string
  readonly email: Option.Option<string>
  readonly address: Option.Option<Address>
}

// ---cut---
const user: User = {
  id: 1,
  username: "john_doe",
  email: Option.some("john.doe@example.com"),
  address: Option.some({
    city: "New York",
    street: Option.some("123 Main St")
  })
}

const street = user.address.pipe(Option.flatMap((address) => address.street))
```

Here's how it works: if the `address` is `Some`, meaning it has a value, the mapping function `(addr) => addr.street` is applied to retrieve the `street` value. On the other hand, if the `address` is `None`, indicating the absence of a value, the mapping function is not executed, and the result is also `None`.

filter

The `Option.filter` function is used to filter an `Option` using a predicate. If the predicate is not satisfied or the `Option` is `None`, it returns `None`.

Let's see an example where we refactor some code to a more idiomatic version:

Original Code

```
import { Option } from "effect"

const removeEmptyString = (input: Option.Option<string>) => {
  if (Option.isSome(input) && input.value === "") {
    return Option.none()
  }
  return input
}

console.log(removeEmptyString(Option.none())) // { _id: 'Option', _tag: 'None' }
console.log(removeEmptyString(Option.some(""))) // { _id: 'Option', _tag: 'None' }
console.log(removeEmptyString(Option.some("a"))) // { _id: 'Option', _tag: 'Some', value: 'a' }
```

Idiomatic Code

```
import { Option } from "effect"

const removeEmptyString = (input: Option.Option<string>) =>
  Option.filter(input, (value) => value !== "")
```

Getting the Value from an Option

To retrieve the value stored within an `Option`, you can use various functions provided by the `Option` module. Let's explore these functions:

- `getOrThrow`: It retrieves the wrapped value from an `Option`, or throws an error if the `Option` is a `None`. Here's an example:

  ```
  import { Option } from "effect"

  Option.getOrThrow(Option.some(10)) // 10
  Option.getOrThrow(Option.none()) // throws getOrThrow called on a None
  ```

- `getOrNull` and `getOrUndefined`: These functions are useful when you want to work with code that doesn't use `Option`. They allow you to retrieve the value of an `Option` as `null` or `undefined`, respectively. Examples:

  ```
  import { Option } from "effect"

  Option.getOrNull(Option.some(5)) // 5
  ```

```
Option.getOrNull(Option.none()) // null

Option.getOrUndefined(Option.some(5)) // 5
Option.getOrUndefined(Option.none()) // undefined
```

- `getOrElse`: This function lets you provide a default value that will be returned if the `Option` is a `None`. Here's an example:

```
import { Option } from "effect"

Option.getOrElse(Option.some(5), () => 0) // 5
Option.getOrElse(Option.none(), () => 0) // 0
```

Fallback

In certain situations, when a computation returns `None`, you may want to try an alternative computation that returns an `Option`. This is where the `Option.orElse` function comes in handy. It allows you to chain multiple computations together and continue with the next one if the previous one resulted in `None`. This can be useful for implementing retry logic, where you want to attempt a computation multiple times until you either succeed or exhaust all possible attempts.

```
import { Option } from "effect"

// Simulating a computation that may or may not produce a result
const performComputation = (): Option.Option<number> =>
  Math.random() < 0.5 ? Option.some(10) : Option.none()

const performAlternativeComputation = (): Option.Option<number> =>
  Math.random() < 0.5 ? Option.some(20) : Option.none()

const result = performComputation().pipe(
  Option.orElse(() => performAlternativeComputation())
)

Option.match(result, {
  onNone: () => console.log("Both computations resulted in None"),
  onSome: (value) => console.log("Computed value:", value) // At least one computation succeeded
})
```

Additionally, the `Option.firstSomeOf` function can be used to retrieve the first value that is `Some` within an iterable of `Option` values:

```
import { Option } from "effect"

const first = Option.firstSomeOf([
  Option.none(),
  Option.some(2),
  Option.none(),
  Option.some(3)
]) // some(2)
```

Interop with Nullable Types

When working with the `Option` data type, you may come across code that uses `undefined` or `null` to represent optional values. The `Option` data type provides several APIs to facilitate the interaction between `Option` and nullable types.

You can create an `Option` from a nullable value using the `fromNullable` API.

```
import { Option } from "effect"

Option.fromNullable(null) // none()
Option.fromNullable(undefined) // none()
Option.fromNullable(1) // some(1)
```

Conversely, if you have a value of type `Option` and want to convert it to a nullable value, you have two options:

- Convert `None` to `null` using the `getOrNull` API.
- Convert `None` to `undefined` using the `getOrUndefined` API.

```
import { Option } from "effect"

Option.getOrNull(Option.some(5)) // 5
Option.getOrNull(Option.none()) // null

Option.getOrUndefined(Option.some(5)) // 5
Option.getOrUndefined(Option.none()) // undefined
```

Interop with Effect

The `Option` type is a subtype of the `Effect` type, which means that it can be seamlessly used with functions from the `Effect` module. These functions are primarily designed to work with `Effect` values, but they can also handle `Option` values and process them correctly.

In the context of `Effect`, the two members of the `Option` type are treated as follows:

- `None` is equivalent to `Effect<never, NoSuchElementException>`

- `Some<A>` is equivalent to `Effect<A>`

To illustrate this interoperability, let's consider the following example:

```
import { Effect, Option } from "effect"

const head = <A>(as: ReadonlyArray<A>): Option.Option<A> =>
  as.length > 0 ? Option.some(as[0]) : Option.none()

console.log(
  Effect.runSync(Effect.succeed([1, 2, 3]).pipe(Effect.andThen(head)))
) // Output: 1

Effect.runSync(Effect.succeed([]).pipe(Effect.andThen(head))) // throws NoSuchElementException: undefined
```

Combining Two or More Options

The `Option.zipWith` function allows you to combine two `Option` values using a provided function. It creates a new `Option` that holds the combined value of both original `Option` values.

```
import { Option } from "effect"

const maybeName: Option.Option<string> = Option.some("John")
const maybeAge: Option.Option<number> = Option.some(25)

const person = Option.zipWith(maybeName, maybeAge, (name, age) => ({
  name: name.toUpperCase(),
  age
}))

console.log(person)
/*
Output:
{ _id: 'Option', _tag: 'Some', value: { name: 'JOHN', age: 25 } }
*/
```

The `Option.zipWith` function takes three arguments:

- The first `Option` you want to combine
- The second `Option` you want to combine
- A function that takes two arguments, which are the values held by the two `Options`, and returns the combined value

It's important to note that if either of the two `Option` values is `None`, the resulting `Option` will also be `None`:

```
import { Option } from "effect"

const maybeName: Option.Option<string> = Option.some("John")
const maybeAge: Option.Option<number> = Option.none()

const person = Option.zipWith(maybeName, maybeAge, (name, age) => ({
  name: name.toUpperCase(),
  age
}))

console.log(person)
/*
Output:
{ _id: 'Option', _tag: 'None' }
*/
```

If you need to combine two or more `Option`s without transforming the values they hold, you can use `Option.all`, which takes a collection of `Option`s and returns an `Option` with the same structure.

- If a tuple is provided, the returned `Option` will contain a tuple with the same length.
- If a struct is provided, the returned `Option` will contain a struct with the same keys.
- If an iterable is provided, the returned `Option` will contain an array.

```
import { Option } from "effect"

const maybeName: Option.Option<string> = Option.some("John")
const maybeAge: Option.Option<number> = Option.some(25)

const tuple = Option.all([maybeName, maybeAge])

const struct = Option.all({ name: maybeName, age: maybeAge })
```

gen

Similar to Effect.gen, there's also `Option.gen`, which provides a convenient syntax, akin to async/await, for writing code involving `Option` and using generators.

Let's revisit the previous example, this time using `Option.gen` instead of `Option.zipWith`:

```
import { Option } from "effect"

const maybeName: Option.Option<string> = Option.some("John")
const maybeAge: Option.Option<number> = Option.some(25)

const person = Option.gen(function* () {
  const name = (yield* maybeName).toUpperCase()
  const age = yield* maybeAge
  return { name, age }
})

console.log(person)
/*
Output:
{ _id: 'Option', _tag: 'Some', value: { name: 'JOHN', age: 25 } }
*/
```

Once again, if either of the two `Option` values is `None`, the resulting `Option` will also be `None`:

```
import { Option } from "effect"

const maybeName: Option.Option<string> = Option.some("John")
const maybeAge: Option.Option<number> = Option.none()

const person = Option.gen(function* () {
  const name = (yield* maybeName).toUpperCase()
  const age = yield* maybeAge
  return { name, age }
})

console.log(person)
/*
Output:
{ _id: 'Option', _tag: 'None' }
*/
```

Comparing Option Values with Equivalence

You can compare `Option` values using the `Option.getEquivalence` function. This function lets you define rules for comparing the contents of `Option` types by providing an `Equivalence` for the type of value they might contain.

Example: Checking Equivalence of Optional Numbers

Imagine you have optional numbers and you want to check if they are equivalent. Here's how you can do it:

```
import { Option, Equivalence } from "effect"

const myEquivalence = Option.getEquivalence(Equivalence.number)

console.log(myEquivalence(Option.some(1), Option.some(1))) // Output: true, because both options contain the number 1
console.log(myEquivalence(Option.some(1), Option.some(2))) // Output: false, because the numbers are different
console.log(myEquivalence(Option.some(1), Option.none()))  // Output: false, because one is a number and the other is empty
```

Sorting Option Values with Order

Sorting a collection of `Option` values can be done using the `Option.getOrder` function. This function helps you sort `Option` values by providing a custom sorting rule for the type of value they might contain.

Example: Sorting Optional Numbers

Suppose you have a list of optional numbers and you want to sort them in ascending order, considering empty values as the lowest:

```
import { Option, Array, Order } from "effect"

const items = [
  Option.some(1),
  Option.none(),
  Option.some(2)
]

const myOrder = Option.getOrder(Order.number)

console.log(Array.sort(myOrder)(items))
/*
Output:
[
  { _id: 'Option', _tag: 'None' },           // None appears first because it's considered the lowest
  { _id: 'Option', _tag: 'Some', value: 1 }, // Sorted in ascending order
  { _id: 'Option', _tag: 'Some', value: 2 }
]
*/
```

In this example, `Option.none()` is treated as the lowest value, allowing `Option.some(1)` and `Option.some(2)` to be sorted in ascending order based on their numerical value. This method ensures that all `Option` values are sorted logically according to their content, with empty values (`Option.none()`) being placed before non-empty values (`Option.some()`).

Advanced Example: Sorting Optional Dates in Reverse Order

Now, let's consider a more complex scenario where you have a list of objects containing optional dates, and you want to sort them in descending order, with any empty optional values placed at the end:

```
import { Option, Array, Order } from "effect"

const items = [
  { data: Option.some(new Date(10)) },
  { data: Option.some(new Date(20)) },
  { data: Option.none() }
]

// Define the order to sort dates within Option values in reverse
const sorted = Array.sortWith(
  items,
  item => item.data,
  Order.reverse(Option.getOrder(Order.Date))
)

console.log(sorted)
/*
Output:
[
  { data: { _id: 'Option', _tag: 'Some', value: '1970-01-01T00:00:00.020Z' } },
  { data: { _id: 'Option', _tag: 'Some', value: '1970-01-01T00:00:00.010Z' } },
  { data: { _id: 'Option', _tag: 'None' } } // None placed last
]
*/
```

Cause

Location: 700-other/300-data-types/cause

Explore the `Cause` data type in the `Effect` type, which stores comprehensive information about failures, including unexpected errors, stack traces, and fiber interruption causes. Learn how `Cause` ensures no failure information is lost, providing a complete story for precise error analysis and handling in your codebase. Discover various causes such as Empty, Fail, Die, Interrupt, Sequential, and Parallel, each representing different error scenarios within the `Effect` workflow.

The Effect<A, E, R> type is polymorphic in values of type `E`, which means we can work with any error type we want. However, there is additional information related to failures that is not captured by the `E` value alone.

To preserve and provide comprehensive information about failures, Effect uses the `Cause<E>` data type. `Cause` is responsible for storing various details, such as:

- Unexpected errors or defects
- Stack and execution traces
- Causes of fiber interruptions

Effect is designed to be strict in preserving all the information related to a failure. It captures and stores the complete story of failure in the `Cause` data type. This ensures that no information about the failure is lost, allowing us to precisely determine what happened during the execution of our effects.

Although we don't typically work directly with `Cause` values, it is an underlying data type that represents errors occurring within an Effect workflow. It provides us with total access to all concurrent and sequential errors within our codebase. This gives us the ability to analyze and handle failures in a comprehensive manner whenever needed.

Creating Causes

We can intentionally create effects with specific causes using the `Effect.failCause` constructor:

```
import { Effect, Cause } from "effect"

// Create an effect that intentionally fails with an empty cause
const effect = Effect.failCause(Cause.empty)
```

To uncover the underlying cause of an effect, we can use the `Effect.cause` function:

```
Effect.cause(effect).pipe(
  Effect.andThen((cause) => ...)
)
```

Cause Variations

There are several causes for various errors, in this section, we will describe each of these causes.

Empty

The `Empty` cause represents a lack of errors.

Fail

The `Fail` cause represents a `Cause` which failed with an expected error of type `E`.

Die

The `Die` cause represents a `Cause` which failed as a result of a defect, or in other words, an unexpected error.

Interrupt

The `Interrupt` cause represents failure due to `Fiber` interruption, which contains the `FiberId` of the interrupted `Fiber`.

Sequential

The `Sequential` cause represents the composition of two causes which occurred sequentially.

For example, if we perform Effect's analog of `try-finally` (i.e. `Effect.ensuring`), and both the `try` and `finally` blocks fail, we have two errors which occurred sequentially. In these cases, the errors can be represented by the `Sequential` cause.

Parallel

The `Parallel` cause represents the composition of two causes which occurred in parallel.

In Effect programs, it is possible that two operations may be performed in parallel. In these cases, the `Effect` workflow can fail for more than one reason. If both computations fail, then there are actually two errors which occurred in parallel. In these cases, the errors can be represented by the `Parallel` cause.

Guards

To identify the type of a `Cause`, you can use specific guards provided by the `Cause` module:

- `Cause.isEmpty`
- `Cause.isFailType`
- `Cause.isDie`
- `Cause.isInterruptType`
- `Cause.isSequentialType`
- `Cause.isParallelType`

Let's see an example of how you can utilize these guards:

```
import { Cause } from "effect"

const cause = Cause.fail(new Error("my message"))

if (Cause.isFailType(cause)) {
  console.log(cause.error.message) // Output: my message
}
```

By using these guards, you can effectively determine the nature of a `Cause`, enabling you to handle different error scenarios appropriately in your code. Whether it's an empty cause, failure, defect, interruption, sequential composition, or parallel composition, these guards provide a clear way to identify and manage various types of errors.

Pattern Matching

In addition to using guards, you can handle different cases of a `Cause` using the `Cause.match` function. This function allows you to define separate callbacks for each possible case of a `Cause`.

Let's explore how you can use `Cause.match`:

```
import { Cause } from "effect"

const cause = Cause.parallel(
  Cause.fail(new Error("my fail message")),
  Cause.die("my die message")
)

console.log(
  Cause.match(cause, {
    onEmpty: "(empty)",
    onFail: (error) => `(error: ${error.message})`,
    onDie: (defect) => `(defect: ${defect})`,
    onInterrupt: (fiberId) => `(fiberId: ${fiberId})`,
    onSequential: (left, right) =>
      `(onSequential (left: ${left}) (right: ${right}))`,
    onParallel: (left, right) =>
      `(onParallel (left: ${left}) (right: ${right})`
  })
)
```

```
/*
Output:
(onParallel (left: (error: my fail message)) (right: (defect: my die message))
*/
```

Pretty Printing

When working with errors in your code, it's crucial to have clear and readable error messages for effective debugging. The `Cause.pretty` function provides a convenient way to achieve this by generating nicely formatted error messages as strings.

Let's take a look at how you can use `Cause.pretty`:

```
import { Cause, FiberId } from "effect"

console.log(Cause.pretty(Cause.empty)) // All fibers interrupted without errors.
console.log(Cause.pretty(Cause.fail(new Error("my fail message")))) // Error: my fail message
console.log(Cause.pretty(Cause.die("my die message"))) // Error: my die message
console.log(Cause.pretty(Cause.interrupt(FiberId.make(1, 0)))) // All fibers interrupted without errors.
console.log(
  Cause.pretty(Cause.sequential(Cause.fail("fail1"), Cause.fail("fail2")))
)
/*
Output:
Error: fail1
Error: fail2
*/
```

Retrieval of Failures and Defects

If you're specifically interested in obtaining a collection of failures or defects from a `Cause`, you can use the `Cause.failures` and `Cause.defects` functions respectively.

Let's see how you can utilize these functions:

```
import { Cause } from "effect"

const cause = Cause.parallel(
  Cause.fail(new Error("my fail message 1")),
  Cause.sequential(
    Cause.die("my die message"),
    Cause.fail(new Error("my fail message 2"))
  )
)

console.log(Cause.failures(cause))
/*
Output:
{
  _id: 'Chunk',
  values: [
    Error: my fail message 1 ...,
    Error: my fail message 2 ...
  ]
}
*/

console.log(Cause.defects(cause))
/*
Output:
{ _id: 'Chunk', values: [ 'my die message' ] }
*/
```

Referenced Data types

Location: 700-other/300-data-types/index

Referenced Data types

Duration

Location: 700-other/300-data-types/duration

Explore the `Duration` data type in Effect for representing non-negative spans of time. Learn to create durations with different units, including milliseconds, seconds, and minutes. Discover options for creating infinite durations and decoding values. Retrieve duration values in milliseconds or nanoseconds. Compare durations and perform arithmetic operations like addition and multiplication. Master the capabilities of the `Duration` module for efficient time handling in your applications.

The `Duration` data type data type is used to represent specific non-negative spans of time. It is commonly used to represent time intervals or durations in various operations, such as timeouts, delays, or scheduling. The `Duration` type provides a convenient way to work with time units and perform calculations on durations.

Creating Durations

The `Duration` module provides several constructors to create durations of different units. Here are some examples:

```
import { Duration } from "effect"

const duration1 = Duration.millis(100) // Create a duration of 100 milliseconds
const duration2 = Duration.seconds(2) // Create a duration of 2 seconds
const duration3 = Duration.minutes(5) // Create a duration of 5 minutes
```

You can create durations using units such as nanoseconds, microsecond, milliseconds, seconds, minutes, hours, days, and weeks.

If you want to create an infinite duration, you can use `Duration.infinity`:

```
import { Duration } from "effect"

console.log(String(Duration.infinity))
/*
Output:
{
  "_id": "Duration",
  "_tag": "Infinity"
}
*/
```

Another option for creating durations is using the `Duration.decode` helper:

- `number`s are treated as milliseconds
- `bigint`s are treated as nanoseconds
- strings must be formatted as `"${number} ${unit}"`

```
// @target: ES2020
import { Duration } from "effect"

Duration.decode(10n) // same as Duration.nanos(10)
Duration.decode(100) // same as Duration.millis(100)
Duration.decode(Infinity) // same as Duration.infinity

Duration.decode("10 nanos") // same as Duration.nanos(10)
Duration.decode("20 micros") // same as Duration.micros(20)
Duration.decode("100 millis") // same as Duration.millis(100)
Duration.decode("2 seconds") // same as Duration.seconds(2)
Duration.decode("5 minutes") // same as Duration.minutes(5)
Duration.decode("7 hours") // same as Duration.hours(7)
Duration.decode("3 weeks") // same as Duration.weeks(3)
```

Getting the Duration Value

You can retrieve the value of a duration in milliseconds using the `toMillis` function:

```
import { Duration } from "effect"

console.log(Duration.toMillis(Duration.seconds(30))) // Output: 30000
```

You can retrieve the value of a duration in nanoseconds using the `toNanos` function:

```
import { Duration } from "effect"

console.log(Duration.toNanos(Duration.millis(100)))
/*
Output:
{
  _id: "Option",
  _tag: "Some",
  value: 100000000n
}
*/
```

Note that `toNanos` returns an `Option<bigint>` since the duration might be infinite. If you want a `bigint` as a return type, you can use `unsafeToNanos`. However, note that it will throw an error for infinite durations:

```
import { Duration } from "effect"

console.log(Duration.unsafeToNanos(Duration.millis(100))) // Output: 100000000n
console.log(Duration.unsafeToNanos(Duration.infinity)) // throws "Cannot convert infinite duration to nanos"
```

Comparing Durations

You can compare two durations using the following functions:

Function	Description
`lessThan`	returns `true` if the first duration is less than the second

Function	Description
`lessThanOrEqualTo`	returns `true` if the first duration is less than or equal to the second
`greaterThan`	returns `true` if the first duration is greater than the second
`greaterThanOrEqualTo`	returns `true` if the first duration is greater than or equal to the second

```
import { Duration } from "effect"

const duration1 = Duration.seconds(30)
const duration2 = Duration.minutes(1)

console.log(Duration.lessThan(duration1, duration2)) // Output: true
console.log(Duration.lessThanOrEqualTo(duration1, duration2)) // Output: true
console.log(Duration.greaterThan(duration1, duration2)) // Output: false
console.log(Duration.greaterThanOrEqualTo(duration1, duration2)) // Output: false
```

Performing Arithmetic Operations

You can perform arithmetic operations on durations, such as addition and multiplication. Here are some examples:

```
import { Duration } from "effect"

const duration1 = Duration.seconds(30)
const duration2 = Duration.minutes(1)

console.log(String(Duration.sum(duration1, duration2)))
/*
Output:
{
  "_id": "Duration",
  "_tag": "Millis",
  "millis": 90000
}
*/

console.log(String(Duration.times(duration1, 2))) // Output: Duration("60000 millis")
/*
Output:
{
  "_id": "Duration",
  "_tag": "Millis",
  "millis": 60000
}
*/
```

Exit

Location: 700-other/300-data-types/exit

Explore the `Exit` data type in Effect, representing the result of executing an `Effect` workflow. Learn about success and failure states, matching with `Exit.match`, and the conceptual relationship between `Exit` and `Either`. Understand how `Exit` is a subtype of `Effect` and its role in handling results and errors.

An `Exit<A, E>` describes the result of executing an `Effect` workflow.

There are two possible values for an `Exit<A, E>`:

- `Exit.Success` contains a success value of type `A`.
- `Exit.Failure` contains a failure Cause of type `E`.

Matching

To handle the different outcomes of an `Exit`, we can use the `Exit.match` function:

```
import { Effect, Exit } from "effect"

const simulatedSuccess = Effect.runSyncExit(Effect.succeed(1))

Exit.match(simulatedSuccess, {
  onFailure: (cause) =>
    console.error(`Exited with failure state: ${cause._tag}`),
  onSuccess: (value) => console.log(`Exited with success value: ${value}`)
})
// Output: "Exited with success value: 1"

const simulatedFailure = Effect.runSyncExit(Effect.fail("error"))

Exit.match(simulatedFailure, {
  onFailure: (cause) =>
    console.error(`Exited with failure state: ${cause._tag}`),
  onSuccess: (value) => console.log(`Exited with success value: ${value}`)
```

```
})
// Output: "Exited with failure state: Fail"
```

In this example, we first simulate a successful `Effect` execution using `Effect.runSyncExit` and `Effect.succeed`. We then handle the `Exit` using `Exit.match`, where the `onSuccess` callback prints the success value.

Next, we simulate a failure using `Effect.runSyncExit` and `Effect.fail`, and handle the `Exit` again using `Exit.match`, where the `onFailure` callback prints the failure state.

Exit vs Either

An `Exit<A, E>` is conceptually an `Either<A, Cause<E>>`. However, it's important to note that Cause encompasses more states than just the expected error type `E`. It also includes other states such as interruption and defects (unexpected errors), as well as the possibility of combining multiple `Cause` values together.

Exit vs Effect

The `Exit` data type is a subtype of the `Effect` type, which means that an `Exit` is itself an `Effect`. The reason for this is that a result can be considered as a *constant computation*. Technically, `Effect.succeed` is an alias for `Exit.succeed`, and `Effect.fail` is an alias for `Exit.fail` (avoiding conversions between `Exit` and `Effect` is important for performance, as boxing and unboxing have a cost).

Redacted

Location: 700-other/300-data-types/redacted

Redacted

The Redacted module provides functionality for handling sensitive information securely within your application. By using the `Redacted` data type, you can ensure that sensitive values are not accidentally exposed in logs or error messages.

make

This function creates a `Redacted<A>` instance from a given value `A`, securely hiding its content.

```
import { Redacted } from "effect"

// Creating a redacted value
const API_KEY = Redacted.make("1234567890")

console.log(API_KEY) // Output: {}
console.log(String(API_KEY)) // Output: <redacted>
```

value

Retrieves the original value from a `Redacted` instance. Use this function with caution, as it exposes the sensitive data.

```
import { Redacted } from "effect"

const API_KEY = Redacted.make("1234567890")

console.log(Redacted.value(API_KEY)) // Output: "1234567890"
```

unsafeWipe

Erases the underlying value of a `Redacted` instance, rendering it unusable. This function is intended to ensure that sensitive data does not remain in memory longer than necessary.

```
import { Redacted } from "effect"

const API_KEY = Redacted.make("1234567890")

console.log(Redacted.value(API_KEY)) // Output: "1234567890"

Redacted.unsafeWipe(API_KEY)

console.log(Redacted.value(API_KEY)) // throws Error: Unable to get redacted value
```

getEquivalence

Generates an equivalence relation for `Redacted<A>` values based on an equivalence relation for the underlying values `A`. This function is useful for comparing `Redacted` instances without exposing their contents.

```
import { Redacted, Equivalence } from "effect"

const API_KEY1 = Redacted.make("1234567890")
```

```
const API_KEY2 = Redacted.make("1-34567890")
const API_KEY3 = Redacted.make("1234567890")

const equivalence = Redacted.getEquivalence(Equivalence.string)

console.log(equivalence(API_KEY1, API_KEY2)) // Output: false
console.log(equivalence(API_KEY1, API_KEY3)) // Output: true
```

Chunk

Location: 700-other/300-data-types/chunk

Explore the benefits of using `Chunk`, an immutable and high-performance array-like data structure in JavaScript. Learn about its advantages, including immutability for concurrent programming and specialized operations for efficient array manipulations. Discover operations like creating, concatenating, dropping elements, comparing for equality, and converting to a `ReadonlyArray`.

A `Chunk<A>` represents a chunk of values of type `A`. Chunks are usually backed by arrays, but expose a purely functional, safe interface to the underlying elements, and they become lazy on operations that would be costly with arrays, such as repeated concatenation. Like lists and arrays, `Chunk` is an ordered collection.

<Warning> `Chunk` is purpose-built to amoritize the cost of repeated concatenation of arrays. Therefore, for use-cases that **do not** involve repeated concatenation of arrays, the overhead of `Chunk` will result in reduced performance. </Warning>

Why Chunk?

Let's explore the reasons behind using `Chunk`:

- **Immutability**: In JavaScript, there is no built-in immutable data type that can efficiently represent an array. While the `Array` type exists, it provides a mutable interface, which means it can be modified after creation. `Chunk`, on the other hand, is an immutable array-like data structure that ensures the data remains unchanged. Immutability is beneficial in various scenarios, especially when dealing with concurrent programming.
- **High Performance**: `Chunk` not only offers immutability but also delivers high performance. It provides specialized operations for common array manipulations, such as appending a single element or concatenating two `Chunk`s together. These specialized operations are significantly faster than performing the same operations on regular JavaScript arrays.

Operations

Creating

To create an empty `Chunk`, you can use the following code:

```
import { Chunk } from "effect"

const emptyChunk = Chunk.empty()
```

If you want to create a `Chunk` with specific values, you can use the `Chunk.make(...values)` function:

```
import { Chunk } from "effect"

const nonEmptyChunk = Chunk.make(1, 2, 3)
```

Alternatively, you can create a `Chunk` by providing a collection of values. There are multiple ways to achieve this:

- Creating a `Chunk` from a generic `Iterable`:

  ```
  import { Chunk, List } from "effect"

  const fromArray = Chunk.fromIterable([1, 2, 3])

  const fromList = Chunk.fromIterable(List.make(1, 2, 3))
  ```

- Creating a `Chunk` from an `Array`:

  ```
  import { Chunk } from "effect"

  const fromUnsafeArray = Chunk.unsafeFromArray([1, 2, 3])
  ```

`Chunk.fromIterable` makes a chunk by cloning the iterable, which can be an expensive process for large iterables or when making many Chunks. `unsafeFromArray` doesn't do any cloning which can have performance benefits, but this breaks the assumption that Chunk is immutable.

<Warning> It is important to note that using `unsafeFromArray` can potentially lead to unsafe or unexpected behavior if the input array is mutated after conversion. If you want to ensure safety, it is recommended to use `fromIterable`. </Warning>

Concatenating

You can concatenate two Chunks using the `appendAll` function:

```
import { Chunk } from "effect"

const concatenatedChunk = Chunk.appendAll(
  Chunk.make(1, 2),
  Chunk.make("a", "b")
)

console.log(concatenatedChunk)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, "a", "b" ]
}
*/
```

Dropping

To drop elements from the beginning of a `Chunk`, you can use the `drop` function:

```
import { Chunk } from "effect"

const droppedChunk = Chunk.drop(Chunk.make(1, 2, 3, 4), 2) // Drops the first 2 elements from the Chunk
```

Comparing

You can compare two `Chunk`s for equality using the `Equal.equals` function:

```
import { Chunk, Equal } from "effect"

const chunk1 = Chunk.make(1, 2)
const chunk2 = Chunk.make(1, 2, 3)

const areEqual = Equal.equals(chunk1, chunk2)
```

Converting

To convert a `Chunk` to a `ReadonlyArray`, you can use the `toReadonlyArray` function:

```
import { Chunk } from "effect"

const readonlyArray = Chunk.toReadonlyArray(Chunk.make(1, 2, 3))
```

Data

Location: 700-other/300-data-types/data

Explore the Data module in Effect, offering functionalities for defining data types, ensuring value equality, and working with case classes. Learn about the advantages of using `Data.struct`, `Data.tuple`, and `Data.array` for efficient value comparisons. Dive into the concept of case classes, including `case`, `tagged`, `Class`, and `TaggedClass`, providing automated implementations for data types. Discover how to create unions of case classes using `TaggedEnum` for streamlined handling of disjoint unions.

The Data module offers a range of features that make it easier to create and manipulate data structures in your TypeScript applications. It includes functionalities for **defining data types**, ensuring **equality** between data objects, and **hashing** data for efficient comparison.

The module offers APIs tailored for comparing **existing values** of your data types. Alternatively, it provides mechanisms for defining **constructors** for your data types.

Value Equality

If you need to compare **existing values** for equality without the need for explicit implementations, consider using the Data module. It provides convenient APIs that generate default implementations for Equal and Hash, making equality checks a breeze.

struct

In this example, we use the `Data.struct` function to create structured data objects and check their equality using `Equal.equals`.

```
import { Data, Equal } from "effect"

const alice = Data.struct({ name: "Alice", age: 30 })

const bob = Data.struct({ name: "Bob", age: 40 })

console.log(Equal.equals(alice, alice)) // Output: true
console.log(Equal.equals(alice, Data.struct({ name: "Alice", age: 30 }))) // Output: true

console.log(Equal.equals(alice, { name: "Alice", age: 30 })) // Output: false
console.log(Equal.equals(alice, bob)) // Output: false
```

The `Data` module simplifies the process by providing a default implementation for both Equal and Hash, allowing you to focus on comparing values without the need for explicit implementations.

tuple

If you prefer to model your domain with tuples, the `Data.tuple` function has got you covered:

```
import { Data, Equal } from "effect"

const alice = Data.tuple("Alice", 30)

const bob = Data.tuple("Bob", 40)

console.log(Equal.equals(alice, alice)) // Output: true
console.log(Equal.equals(alice, Data.tuple("Alice", 30))) // Output: true

console.log(Equal.equals(alice, ["Alice", 30])) // Output: false
console.log(Equal.equals(alice, bob)) // Output: false
```

array

You can take it a step further and use arrays to compare multiple values:

```
import { Data, Equal } from "effect"

const alice = Data.struct({ name: "Alice", age: 30 })
const bob = Data.struct({ name: "Bob", age: 40 })

const persons = Data.array([alice, bob])

console.log(
  Equal.equals(
    persons,
    Data.array([
      Data.struct({ name: "Alice", age: 30 }),
      Data.struct({ name: "Bob", age: 40 })
    ])
  )
) // Output: true
```

In this extended example, we create an array of person objects using the `Data.array` function. We then compare this array with another array of person objects using `Equal.equals`, and the result is `true` since the arrays contain structurally equal elements.

Constructors

The module introduces a concept known as "Case classes", which automate various essential operations when defining data types. These operations include generating **constructors**, handling **equality** checks, and managing **hashing**.

Case classes can be defined in two primary ways:

- as plain objects using `case` or `tagged`
- as TypeScript classes using `Class` or `TaggedClass`

case

This helper automatically provides implementations for constructors, equality checks, and hashing for your data type.

```
import { Data, Equal } from "effect"

interface Person {
  readonly name: string
}

// Creating a constructor for `Person`
const Person = Data.case<Person>()

// Creating instances of Person
const mike1 = Person({ name: "Mike" })
const mike2 = Person({ name: "Mike" })
const john = Person({ name: "John" })

// Checking equality
console.log(Equal.equals(mike1, mike2)) // Output: true
console.log(Equal.equals(mike1, john)) // Output: false
```

Here we create a constructor for `Person` using `Data.case`. The resulting instances come with built-in equality checks, making it simple to compare them using `Equal.equals`.

tagged

In certain situations, like when you're defining a data type that includes a tag field (commonly used in disjoint unions), using the `case` approach can become repetitive and cumbersome. This is because you're required to specify the tag every time you create an instance:

```
import { Data } from "effect"

interface Person {
  readonly _tag: "Person" // the tag
  readonly name: string
}

const Person = Data.case<Person>()

// It can be quite frustrating to repeat `_tag: 'Person'` every time...
const mike = Person({ _tag: "Person", name: "Mike" })
const john = Person({ _tag: "Person", name: "John" })
```

To make your life easier, the `tagged` helper simplifies this process by allowing you to define the tag only once. It follows the convention within the Effect ecosystem of naming the tag field with `"_tag"`:

```
import { Data } from "effect"

interface Person {
  readonly _tag: "Person" // the tag
  readonly name: string
}

const Person = Data.tagged<Person>("Person")

// Now, it's much more convenient...
const mike = Person({ name: "Mike" })
const john = Person({ name: "John" })

console.log(mike._tag) // Output: { name: 'Mike', _tag: 'Person' }
```

Class

If you find it more comfortable to work with classes instead of plain objects, you have the option to use `Data.Class` instead of `case`. This approach can be particularly useful in scenarios where you prefer a more class-oriented structure:

```
import { Data, Equal } from "effect"

class Person extends Data.Class<{ name: string }> {}

// Creating instances of Person
const mike1 = new Person({ name: "Mike" })
const mike2 = new Person({ name: "Mike" })
const john = new Person({ name: "John" })

// Checking equality
console.log(Equal.equals(mike1, mike2)) // Output: true
console.log(Equal.equals(mike1, john)) // Output: false
```

One advantage of using classes is that you can easily add custom getters and methods to the class definition, enhancing its functionality to suit your specific needs:

```
import { Data } from "effect"

class Person extends Data.Class<{ name: string }> {
  get upperName() {
    return this.name.toUpperCase()
  }
}

const mike = new Person({ name: "Mike" })

console.log(mike.upperName) // Output: MIKE
```

TaggedClass

For those who prefer working with classes over plain objects, you can utilize `Data.TaggedClass` as an alternative to `tagged`.

```
import { Data, Equal } from "effect"

class Person extends Data.TaggedClass("Person")<{ name: string }> {}

// Creating instances of Person
const mike1 = new Person({ name: "Mike" })
const mike2 = new Person({ name: "Mike" })
const john = new Person({ name: "John" })

console.log(mike1) // Output: Person { name: 'Mike', _tag: 'Person' }

// Checking equality
console.log(Equal.equals(mike1, mike2)) // Output: true
console.log(Equal.equals(mike1, john)) // Output: false
```

One of the advantages of using tagged classes is that you can seamlessly incorporate custom getters and methods into the class definition, expanding its functionality as needed:

```
import { Data } from "effect"

class Person extends Data.TaggedClass("Person")<{ name: string }> {
  get upperName() {
    return this.name.toUpperCase()
  }
}

const mike = new Person({ name: "Mike" })

console.log(mike.upperName) // Output: MIKE
```

Union of Tagged Structs

If you're looking to create a disjoint union of tagged structs, you can easily achieve this using `Data.TaggedEnum` and `Data.taggedEnum`. This feature simplifies the process of defining and working with unions of plain objects.

Definition

Let's walk through an example to see how this works:

```
import { Data, Equal } from "effect"

// Define a union type using TaggedEnum
type RemoteData = Data.TaggedEnum<{
  Loading: {}
  Success: { readonly data: string }
  Failure: { readonly reason: string }
}>

// Create constructors for specific error types
const { Loading, Success, Failure } = Data.taggedEnum<RemoteData>()

// Create instances of errors
const state1 = Loading()
const state2 = Success({ data: "test" })
const state3 = Success({ data: "test" })
const state4 = Failure({ reason: "not found" })

// Checking equality
console.log(Equal.equals(state2, state3)) // Output: true
console.log(Equal.equals(state2, state4)) // Output: false

console.log(state1) // Output: { _tag: 'Loading' }
console.log(state2) // Output: { data: 'test', _tag: 'Success' }
console.log(state4) // Output: { reason: 'not found', _tag: 'Failure' }
```

In this example:

- We define a `RemoteData` union type with three states: `Loading`, `Success`, and `Failure`.
- We use `Data.taggedEnum` to create constructors for these states.
- We create instances of each state and check for equality using `Equal.equals`.

Note that it follows the convention within the Effect ecosystem of naming the tag field with `"_tag"`.

Adding Generics

You can also create tagged unions with generics using `TaggedEnum.WithGenerics`. This allows for more flexible and reusable type definitions.

```
import { Data } from "effect"

type RemoteData<Success, Failure> = Data.TaggedEnum<{
  Loading: {}
  Success: { data: Success }
  Failure: { reason: Failure }
}>

interface RemoteDataDefinition extends Data.TaggedEnum.WithGenerics<2> {
  readonly taggedEnum: RemoteData<this["A"], this["B"]>
}

const { Loading, Failure, Success } = Data.taggedEnum<RemoteDataDefinition>()

const loading = Loading()

const failure = Failure({ reason: "not found" })

const success = Success({ data: 1 })
```

$is and $match

The `Data.taggedEnum` also provides `$is` and `$match` functions for type guards and pattern matching, respectively.

```
import { Data } from "effect"

type RemoteData = Data.TaggedEnum<{
  Loading: {}
  Success: { readonly data: string }
  Failure: { readonly reason: string }
}>

const { $is, $match, Loading, Success, Failure } =
  Data.taggedEnum<RemoteData>()

// Create a type guard
const isLoading = $is("Loading")

console.log(isLoading(Loading())) // true
console.log(isLoading(Success({ data: "test" }))) // false

// Create a matcher
const matcher = $match({
  Loading: () => "this is a Loading",
  Success: ({ data }) => `this is a Success: ${data}`,
  Failure: ({ reason }) => `this is a Failre: ${reason}`
})

console.log(matcher(Success({ data: "test" }))) // "this is a Success: test"
```

Errors

In Effect, errors play a crucial role, and defining and constructing them is made easier with two specialized constructors:

- `Error`
- `TaggedError`

Error

With `Data.Error`, we can create an `Error` with additional fields beyond the usual `message`:

```
import { Data } from "effect"

class NotFound extends Data.Error<{ message: string; file: string }> {}

const err = new NotFound({
  message: "Cannot find this file",
  file: "foo.txt"
})

console.log(err instanceof Error) // Output: true

console.log(err.file) // Output: foo.txt
console.log(err)
/*
Output:
Error: Cannot find this file
  ... stack trace ...
*/
```

Additionally, `NotFound` is "yieldable" as it is an `Effect`, so there's no need to use `Effect.fail`:

```
import { Data, Effect } from "effect"

class NotFound extends Data.Error<{ message: string; file: string }> {}

const program = Effect.gen(function* () {
  yield* new NotFound({
    message: "Cannot find this file",
    file: "foo.txt"
  })
})
```

TaggedError

In Effect, there's a special convention to add a `_tag` field to custom errors. This convention simplifies certain operations, such as error handling with APIs like `Effect.catchTag` or `Effect.catchTags`. Therefore, the `TaggedError` API simplifies the process of creating custom errors by automatically adding this type of tag without needing to specify it every time you create a new error:

```
import { Data, Effect, Console } from "effect"

class NotFound extends Data.TaggedError("NotFound")<{
  message: string
  file: string
}> {}

const program = Effect.gen(function* () {
  yield* new NotFound({
    message: "Cannot find this file",
```

```
    file: "foo.txt"
  })
}).pipe(
  Effect.catchTag("NotFound", (err) =>
    Console.error(`${err.message} (${err.file})`)
  )
)

Effect.runPromise(program)
// Output: Cannot find this file (foo.txt)
```

Effect vs Promise

Location: 700-other/900-effect-vs-promise

Explore the differences between `Promise` and `Effect` in TypeScript, covering type safety, creation, chaining, and concurrency. Learn how `Effect` enhances type tracking for errors and dependencies and provides powerful features like fiber-based concurrency and built-in capabilities for logging, scheduling, caching, and more.

In this guide, we will explore the differences between `Promise` and `Effect`, two approaches to handling asynchronous operations in TypeScript. We'll discuss their type safety, creation, chaining, and concurrency, providing examples to help you understand their usage.

Comparing Effects and Promises: Key Distinctions

- **Evaluation Strategy:** Promises are eagerly evaluated, whereas effects are lazily evaluated.
- **Execution Mode:** Promises are one-shot, executing once, while effects are multi-shot, repeatable.
- **Interruption Handling and Automatic Propagation:** Promises lack built-in interruption handling, posing challenges in managing interruptions, and don't automatically propagate interruptions, requiring manual abort controller management. In contrast, effects come with interruption handling capabilities and automatically compose interruption, simplifying management locally on smaller computations without the need for high-level orchestration.
- **Structured Concurrency:** Effects offer structured concurrency built-in, which is challenging to achieve with Promises.
- **Error Reporting (Type Safety):** Promises don't inherently provide detailed error reporting at the type level, whereas effects do, offering type-safe insight into error cases.
- **Runtime Behavior:** The Effect runtime aims to remain synchronous as long as possible, transitioning into asynchronous mode only when necessary due to computation requirements or main thread starvation.

Type safety

Let's start by comparing the types of `Promise` and `Effect`. The type parameter `A` represents the resolved value of the operation:

Here's what sets `Effect` apart:

- It allows you to track the types of errors statically through the type parameter `Error`. For more information about error management in `Effect`, see Expected Errors.
- It allows you to track the types of required dependencies statically through the type parameter `Context`. For more information about context management in `Effect`, see Managing Services.

Creating

Success

Let's compare creating a successful operation using `Promise` and `Effect`:

Failure

Now, let's see how to handle failures with `Promise` and `Effect`:

Constructor

Creating operations with custom logic:

Thenable

Mapping the result of an operation:

map

flatMap

Chaining multiple operations:

Comparing Effect.gen with async/await

If you are familiar with `async`/`await`, you may notice that the flow of writing code is similar.

Let's compare the two approaches:

It's important to note that although the code appears similar, the two programs are not identical. The purpose of comparing them side by side is just to highlight the resemblance in how they are written.

Concurrency

Promise.all()

Promise.allSettled()

Promise.any()

Promise.race()

FAQ

Question. What is the equivalent of starting a promise without immediately waiting for it in Effects?

```
const task = (delay: number, name: string) =>
  new Promise((resolve) =>
    setTimeout(() => {
      console.log(`${name} done`)
      return resolve(name)
    }, delay)
  )

export async function program() {
  const r0 = task(2_000, "long running task")
  const r1 = await task(200, "task 2")
  const r2 = await task(100, "task 3")
  return {
    r1,
    r2,
    r0: await r0
  }
}

program().then(console.log)
/*
Output:
task 2 done
task 3 done
long running task done
{ r1: 'task 2', r2: 'task 3', r0: 'long running promise' }
*/
```

Answer: You can achieve this by utilizing `Effect.fork` and `Fiber.join`.

```
import { Effect, Fiber } from "effect"

const task = (delay: number, name: string) =>
  Effect.gen(function* () {
    yield* Effect.sleep(delay)
    console.log(`${name} done`)
    return name
  })

const program = Effect.gen(function* () {
  const r0 = yield* Effect.fork(task(2_000, "long running task"))
  const r1 = yield* task(200, "task 2")
  const r2 = yield* task(100, "task 3")
  return {
    r1,
    r2,
    r0: yield* Fiber.join(r0)
  }
})

Effect.runPromise(program).then(console.log)
/*
Output:
task 2 done
task 3 done
long running task done
{ r1: 'task 2', r2: 'task 3', r0: 'long running promise' }
*/
```

Myths

Location: 700-other/150-myths

Discuss common misconceptions about Effect.

Effect heavily relies on generators and generators are slow!

Effect's internals are not built on generators, we only use generators to provide an API which closely mimics async-await. Internally async-await uses the same mechanics as generators and they are equally performant. So if you don't have a problem with async-await you won't have a problem with Effect's generators.

Where generators and iterables are unacceptably slow is in transforming collections of data, for that try to use plain arrays as much as possible.

Effect will make your code 500x slower!

Effect does perform 500x slower if you are comparing:

```
const result = 1 + 1
```

to

```
const result = Effect.runSync(Effect.zipWith(
  Effect.succeed(1),
  Effect.succeed(1),
  (a, b) => a + b
))
```

The reason is one operation is optimized by the JIT compiler to be a direct CPU instruction and the other isn't.

In reality you'd never use Effect in such cases, Effect is an app-level library to tame concurrency, error handling, and much more!

You'd use Effect to coordinate your thunks of code, and you can build your thunks of code in the best perfoming manner as you see fit while still controlling execution through Effect.

Effect has a huge performance overhead!

Depends what you mean by performance, many times performance bottlenecks in JS are due to bad management of concurrency.

Thanks to structured concurrency and observability it becomes much easier to spot and optimize those issues.

There are apps in frontend running at 120fps that use Effect intensively, so most likely effect won't be your perf problem.

In regards of memory, it doesn't use much more memory than a normal program would, there are a few more allocations compared to non Effect code but usually this is no longer the case when the non Effect code does the same thing as the Effect code.

The advise would be start using it and monitor your code, optimise out of need not out of thought, optimizing too early is the root of all evils in software design.

The bundle size is HUGE!

Effect's minimum cost is about 25k of gzipped code, that chunk contains the Effect Runtime and already includes almost all the functions that you'll need in a normal app-code scenario.

From that point on Effect is tree-shaking friendly so you'll only include what you use.

Also when using Effect your own code becomes shorter and terser, so the overall cost is amortized with usage, we have apps where adopting Effect in the majority of the codebase led to reduction of the final bundle.

Effect is impossible to learn, there are so many functions and modules!

True, the full Effect ecosystem is quite large and some modules contain 1000s of functions, the reality is that you don't need to know them all to start being productive, you can safely start using Effect knowing just 10-20 functions and progressively discover the rest, just like you can start using TypeScript without knowing every single NPM package.

A short list of commonly used functions to begin are:

- Effect.succeed
- Effect.fail
- Effect.sync
- Effect.tryPromise
- Effect.gen
- Effect.runPromise
- Effect.catchTag

- Effect.catchAll
- Effect.acquireRelease
- Effect.acquireUseRelease
- Effect.provide
- Effect.provideService
- Effect.andThen
- Effect.map
- Effect.tap

A short list of commonly used modules:

- Effect
- Context
- Layer
- Option
- Either
- Array
- Match

Effect is the same as RxJS and shares its problems

This is a sensitive topic, let's start by saying that RxJS is a great project and that it has helped millions of developers write reliable software and we all should be thankful to the developers who contributed to such an amazing project.

Discussing the scope of the projects, RxJS aims to make working with Observables easy and wants to provide reactive extensions to JS, Effect instead wants to make writing production-grade TypeScript easy. While the intersection is non-empty the projects have fundamentally different objectives and strategies.

Sometimes people refer to RxJS in bad light, and the reason isn't RxJS in itself but rather usage of RxJS in problem domains where RxJS wasn't thought to be used.

Namely the idea that "everything is a stream" is theoretically true but it leads to fundamental limitations on developer experience, the primary issue being that streams are multi-shot (emit potentially multiple elements, or zero) and mutable delimited continuations (JS Generators) are known to be only good to represent single-shot effects (that emit a single value).

In short it means that writing in imperative style (think of async/await) is practically impossible with stream primitives (practically because there would be the option of replaying the generator at every element and at every step, but this tends to be inefficient and the semantics of it are counter-intuitive, it would only work under the assumption that the full body is free of side-effects), forcing the developer to use declarative approaches such as pipe to represent all of their code.

Effect has a Stream module (which is pull-based instead of push-based in order to be memory constant), but the basic Effect type is single-shot and it is optimised to act as a smart & lazy Promise that enables imperative programming, so when using Effect you're not forced to use a declarative style for everything and you can program using a model which is similar to async-await.

The other big difference is that RxJS only cares about the happy-path with explicit types, it doesn't offer a way of typing errors and dependencies, Effect instead consider both errors and dependencies as explicitely typed and offers control-flow around those in a fully type-safe manner.

In short if you need reactive programming around Observables, use RxJS, if you need to write production-grade TypeScript that includes by default native telemetry, error handling, dependency injection, and more use Effect.

Effect should be a language or Use a different language

Neither solve the issue of writing production grade software in TypeScript.

TypeScript is an amazing language to write full stack code with deep roots in the JS ecosystem and wide compatibility of tools, it is an industrial language adopted by many large scale companies.

The fact that something like Effect is possible within the language and the fact that the language supports things such as generators that allows for imperative programming with custom types such as Effect makes TypeScript a unique language.

In fact even in functional languages such as Scala the interop with effect systems is less optimal than it is in TypeScript, to the point that effect system authors have expressed wish for their language to support as much as TypeScript supports.

Equal

Location: 700-other/400-trait/equal

The Equal module provides a solution for value-based equality checks, addressing issues with JavaScript's native reference-based equality operators. Developers can define custom equality checks, ensuring data integrity and promoting predictable behavior. To implement custom equality, developers can either implement the `Equal` interface or leverage the simpler solution offered by the [Data](../data-types/data) module, which automatically generates default implementations for both `Equal` and `Hash`. This excerpt also explores working with collections like `HashSet` and `HashMap` to handle value-based equality checks effectively.

The Equal module provides a simple and convenient way to define and check for equality between two values in TypeScript.

Here are some key reasons why Effect exports an Equal module:

1. **Value-Based Equality**: JavaScript's native equality operators (=== and ==) check for equality by reference, meaning they compare objects based on their memory addresses rather than their content. This behavior can be problematic when you want to compare objects with the same values but different references. The Equal module offers a solution by allowing developers to define custom equality checks based on the values of objects.

2. **Custom Equality**: The Equal module enables developers to implement custom equality checks for their data types and classes. This is crucial when you have specific requirements for determining when two objects should be considered equal. By implementing the `Equal` interface, developers can define their own equality logic.

3. **Data Integrity**: In some applications, maintaining data integrity is crucial. The ability to perform value-based equality checks ensures that identical data is not duplicated within collections like sets or maps. This can lead to more efficient memory usage and more predictable behavior.

4. **Predictable Behavior**: The Equal module promotes more predictable behavior when comparing objects. By explicitly defining equality criteria, developers can avoid unexpected results that may occur with JavaScript's default reference-based equality checks.

How to Perform Equality Checking in Effect

In Effect it's advisable to **stop using** JavaScript's `===` and `==` operators and instead rely on the `Equal.equals` function. This function can work with any data type that implements the `Equal` trait. Some examples of such data types include Option, Either, HashSet, and HashMap.

When you use `Equal.equals` and your objects do not implement the `Equal` trait, it defaults to using the `===` operator for object comparison:

```
import { Equal } from "effect"

const a = { name: "Alice", age: 30 }
const b = { name: "Alice", age: 30 }

console.log(Equal.equals(a, b)) // Output: false
```

In this example, `a` and `b` are two separate objects with the same contents. However, `===` considers them different because they occupy different memory locations. This behavior can lead to unexpected results when you want to compare values based on their content.

However, you can configure your models to ensure that `Equal.equals` behaves consistently with your custom equality checks. There are two alternative approaches:

1. **Implementing the `Equal` Interface**: This method is useful when you need to define your custom equality check.

2. **Using the Data Module**: For simple value equality, the Data module provides a more straightforward solution by automatically generating default implementations for `Equal`.

Let's delve into both solutions.

Implementing the Equal Interface

To create custom equality behavior, you can implement the `Equal` interface in your models. This interface extends the `Hash` interface from the Hash module.

Here's an example of implementing the `Equal` interface for a `Person` class:

```
import { Equal, Hash } from "effect"

export class Person implements Equal.Equal {
  constructor(
    readonly id: number,  // Unique identifier for each person
    readonly name: string,
    readonly age: number
  ) {}

  // Defines equality based on id, name, and age
  [Equal.symbol](that: Equal.Equal): boolean {
    if (that instanceof Person) {
      return (
        Equal.equals(this.id, that.id) &&
        Equal.equals(this.name, that.name) &&
        Equal.equals(this.age, that.age)
      )
    }
    return false
  }

  // Generates a hash code based primarily on the unique id
  [Hash.symbol](): number {
    return Hash.hash(this.id)
  }
}

// @include: Person
```

In the above code, we define a custom equality function `[Equal.symbol]` and a hash function `[Hash.symbol]` for the `Person` class. The `Hash` interface optimizes equality checks by comparing hash values instead of the objects themselves. When you use the `Equal.equals` function to compare two objects, it first checks if their hash values are equal. If not, it quickly determines that the objects are not equal, avoiding the need for a detailed property-by-property comparison.

Once you've implemented the `Equal` interface, you can utilize the `Equal.equals` function to check for equality using your custom logic. Here's an example using the `Person` class:

```
// @filename: Person.ts
// @include: Person

// @filename: index.ts
// ---cut---
import { Equal } from "effect"
import { Person } from "./Person"

const alice = new Person(1, "Alice", 30)
console.log(Equal.equals(alice, new Person(1, "Alice", 30))) // Output: true

const bob = new Person(2, "Bob", 40)
console.log(Equal.equals(alice, bob)) // Output: false
```

In this code, the equality check returns `true` when comparing `alice` to a new `Person` object with identical property values and `false` when comparing `alice` to `bob` due to their differing property values.

Simplifying Equality with the Data Module

Implementing both `Equal` and `Hash` can become cumbersome when all you need is straightforward value equality checks. Luckily, the Data module provides a simpler solution. It offers APIs that automatically generate default implementations for both `Equal` and `Hash`.

Let's see how it works:

```
import { Equal, Data } from "effect"

const alice = Data.struct({ name: "Alice", age: 30 })

const bob = Data.struct({ name: "Bob", age: 40 })

console.log(Equal.equals(alice, alice)) // Output: true
console.log(Equal.equals(alice, Data.struct({ name: "Alice", age: 30 }))) // Output: true

console.log(Equal.equals(alice, { name: "Alice", age: 30 })) // Output: false
console.log(Equal.equals(alice, bob)) // Output: false
```

In this example, we use the `Data.struct` function to create structured data objects and check their equality using `Equal.equals`. The Data module simplifies the process by providing a default implementation for both `Equal` and `Hash`, allowing you to focus on comparing values without the need for explicit implementations.

The Data module isn't limited to just structs. It can handle various data types, including tuples, arrays, and records. If you're curious about how to leverage its full range of features, you can explore the Data module documentation.

Working with Collections

JavaScript's built-in `Set` and `Map` can be a bit tricky when it comes to checking equality:

```
export const set = new Set()

set.add({ name: "Alice", age: 30 })
set.add({ name: "Alice", age: 30 })

console.log(set.size) // Output: 2
```

Even though the two elements in the set have the same values, the set contains two elements. Why? JavaScript's `Set` checks for equality by reference, not by values.

To perform value-based equality checks, you'll need to use the `Hash*` collection types available in the `effect` package. These collection types, such as HashSet and HashMap, provide support for the `Equal` trait.

Let's take a closer look at how to use `HashSet` for value-based equality checks:

```
import { HashSet, Data } from "effect"

const set = HashSet.empty().pipe(
  HashSet.add(Data.struct({ name: "Alice", age: 30 })),
  HashSet.add(Data.struct({ name: "Alice", age: 30 }))
)

console.log(HashSet.size(set)) // Output: 1
```

When you use the `HashSet`, it correctly handles value-based equality checks. In this example, even though you're adding two objects with the same values, the `HashSet` treats them as a single element.

Note: It's crucial to use elements that implement the `Equal` trait, either by implementing custom equality checks or by using the Data module. This ensures proper functionality when working with `HashSet`. Without this, you'll encounter the same behavior as the native `Set` data type:

```
import { HashSet } from "effect"

const set = HashSet.empty().pipe(
  HashSet.add({ name: "Alice", age: 30 }),
  HashSet.add({ name: "Alice", age: 30 })
)

console.log(HashSet.size(set)) // Output: 2
```

In this case, without using the Data module alongside `HashSet`, you'll experience the same behavior as the native `Set` data type. The set contains two elements because it checks for equality by reference, not by values.

When working with the `HashMap`, you have the advantage of comparing keys by their values instead of their references. This is particularly helpful in scenarios where you want to associate values with keys based on their content.

Let's explore this concept with a practical example:

```
import { HashMap, Data } from "effect"

const map = HashMap.empty().pipe(
  HashMap.set(Data.struct({ name: "Alice", age: 30 }), 1),
  HashMap.set(Data.struct({ name: "Alice", age: 30 }), 2)
)

console.log(HashMap.size(map)) // Output: 1

console.log(HashMap.get(map, Data.struct({ name: "Alice", age: 30 })))
/*
Output:
{
  _id: "Option",
  _tag: "Some",
  value: 2
}
*/
```

In this code snippet, we use the `HashMap` data structure to create a map where keys are objects created using `Data.struct`. These objects have the same values, which would typically result in multiple entries in a traditional JavaScript map.

However, with `HashMap`, the keys are compared by their values rather than their memory references. As a result, even though we add two objects with identical content as keys, the map correctly handles them as a single key-value pair.

To retrieve a value associated with a specific key, we can use `HashMap.get`. In this example, when we query the map with an object having the same values as the key, it returns the associated value, which is `2`.

Traits

Location: 700-other/400-trait/index

Traits

Hash

Location: 700-other/400-trait/hash

Hash trait documentation

The `Hash` trait in Effect is closely tied to the Equal trait and serves a supportive role in optimizing equality checks by providing a mechanism for hashing. Hashing is a crucial step in the efficient determination of equality between two values, particularly when used with data structures like hash tables.

Role of Hash in Equality Checking

The main function of the `Hash` trait is to provide a quick and efficient way to determine if two values are definitely not equal, thereby complementing the Equal trait. When two values implement the Equal trait, their hash values (computed using the `Hash` trait) are compared first:

- **Different Hash Values**: If the hash values are different, it is guaranteed that the values themselves are different. This quick check allows the system to avoid a potentially expensive equality check.
- **Same Hash Values**: If the hash values are the same, it does not guarantee that the values are equal, only that they might be. In this case, a more thorough comparison using the Equal trait is performed to determine actual equality.

This method dramatically speeds up the equality checking process, especially in collections where quick look-up and insertion times are crucial, such as in hash sets or hash maps.

Practical Example and Explanation

Consider a scenario where you have a custom `Person` class, and you want to check if two instances are equal based on their properties. By implementing both the `Equal` and `Hash` traits, you can efficiently manage these checks:

```
import { Equal, Hash } from "effect"

class Person implements Equal.Equal {
  constructor(
    readonly id: number,  // Unique identifier for each person
    readonly name: string,
    readonly age: number
  ) {}

  // Defines equality based on id, name, and age
  [Equal.symbol](that: Equal.Equal): boolean {
    if (that instanceof Person) {
      return (
        Equal.equals(this.id, that.id) &&
        Equal.equals(this.name, that.name) &&
        Equal.equals(this.age, that.age)
      )
    }
    return false
  }

  // Generates a hash code based primarily on the unique id
  [Hash.symbol](): number {
    return Hash.hash(this.id)
  }
}

const alice = new Person(1, "Alice", 30)
console.log(Equal.equals(alice, new Person(1, "Alice", 30))) // Output: true

const bob = new Person(2, "Bob", 40)
console.log(Equal.equals(alice, bob)) // Output: false
```

In this code snippet:

- The `[Equal.symbol]` method determines equality by comparing the `id`, `name`, and `age` fields of `Person` instances. This approach ensures that the equality check is comprehensive and considers all relevant attributes.
- The `[Hash.symbol]` method computes a hash code using the `id` of the person. This value is used to quickly differentiate between instances in hashing operations, optimizing the performance of data structures that utilize hashing.
- The equality check returns `true` when comparing `alice` to a new `Person` object with identical property values and `false` when comparing `alice` to `bob` due to their differing property values.

Effect vs fp-ts

Location: 700-other/800-fp-ts

A detailed comparison of key features between the Effect and fp-ts libraries, including typed services, built-in services, error handling, pipeable APIs, dual APIs, testability, resource management, interruptions, defects, fiber-based concurrency, retry policies, logging, scheduling, caching, batching, metrics, tracing, configuration, immutable data structures, and stream processing.

FAQ

Bundle Size Comparison Between Effect and fp-ts

Q: I compared the bundle sizes of two simple programs using Effect and fp-ts. Why does Effect have a larger bundle size?

A: It's natural to observe different bundle sizes because Effect and fp-ts are distinct systems designed for different purposes. Effect's bundle size is larger due to its included fiber runtime, which is crucial for its functionality. While the initial bundle size may seem large, the overhead amortizes as you use Effect.

Q: Should I be concerned about the bundle size difference when choosing between Effect and fp-ts?

A: Not necessarily. Consider the specific requirements and benefits of each library for your project.

Comparison Table

The following table compares the features of the Effect and fp-ts libraries.

Feature	fp-ts	Effect
Typed Services	❌	✅
Built-in Services	❌	✅

Feature	fp-ts	Effect
Typed errors	✅	✅
Pipeable APIs	✅	✅
Dual APIs	❌	✅
Testability	❌	✅
Resource Management	❌	✅
Interruptions	❌	✅
Defects	❌	✅
Fiber-Based Concurrency	❌	✅
Fiber Supervision	❌	✅
Retry and Retry Policies	❌	✅
Built-in Logging	❌	✅
Built-in Scheduling	❌	✅
Built-in Caching	❌	✅
Built-in Batching	❌	✅
Metrics	❌	✅
Tracing	❌	✅
Configuration	❌	✅
Immutable Data Structures	❌	✅
Stream Processing	❌	✅

Here's an explanation of each feature:

Typed Services

Both fp-ts and Effect libraries provide the ability to track requirements at the type level, allowing you to define and use services with specific types. In fp-ts, you can utilize the `Reader<R, E, A>` type, while in Effect, the `Effect<A, E, R>` type is available. It's important to note that in fp-ts, the `R` type parameter is contravariant, which means that there is no guarantee of avoiding conflicts, and the library offers only basic tools for dependency management.

On the other hand, in Effect, the `R` type parameter is covariant and all APIs have the ability to merge dependencies at the type level when multiple effects are involved. Effect also provides a range of specifically designed tools to simplify the management of dependencies, including `Tag`, `Context`, and `Layer`. These tools enhance the ease and flexibility of handling dependencies in your code, making it easier to compose and manage complex applications.

Built-in Services

The Effect library has built-in services like `Clock`, `Random` and `Tracer`, while fp-ts does not provide any default services.

Typed errors

Both libraries support typed errors, enabling you to define and handle errors with specific types. However, in Effect, all APIs have the ability to merge errors at the type-level when multiple effects are involved, and each effect can potentially fail with different types of errors.

This means that when combining multiple effects that can fail, the resulting type of the error will be a union of the individual error types. Effect provides utilities and type-level operations to handle and manage these merged error types effectively.

Pipeable APIs

Both fp-ts and Effect libraries provide pipeable APIs, allowing you to compose and sequence operations in a functional and readable manner using the `pipe` function. However, Effect goes a step further and offers a `.pipe()` method on each data type, making it more convenient to work with pipelines without the need to explicitly import the `pipe` function every time.

Dual APIs

Effect library provides dual APIs, allowing you to use the same API in different ways (e.g., "data-last" and "data-first" variants).

Testability

The functional style of fp-ts generally promotes good testability of the code written using it, but the library itself does not provide dedicated tools specifically designed for the testing phase. On the other hand, Effect takes testability a step further by offering additional tools that are specifically

tailored to simplify the testing process.

Effect provides a range of utilities that improve testability. For example, it offers the `TestClock` utility, which allows you to control the passage of time during tests. This is useful for testing time-dependent code. Additionally, Effect provides the `TestRandom` utility, which enables fully deterministic testing of code that involves randomness. This ensures consistent and predictable test results. Another helpful tool is `ConfigProvider.fromMap`, which makes it easy to define mock configurations for your application during testing.

Resource Management

The Effect library provides built-in capabilities for resource management, while fp-ts has limited features in this area (mainly `bracket`) and they are less sophisticated.

In Effect, resource management refers to the ability to acquire and release resources, such as database connections, file handles, or network sockets, in a safe and controlled manner. The library offers comprehensive and refined mechanisms to handle resource acquisition and release, ensuring proper cleanup and preventing resource leaks.

Interruptions

The Effect library supports interruptions, which means you can interrupt and cancel ongoing computations if needed. This feature gives you more control over the execution of your code and allows you to handle situations where you want to stop a computation before it completes.

In Effect, interruptions are useful in scenarios where you need to handle user cancellations, timeouts, or other external events that require stopping ongoing computations. You can explicitly request an interruption and the library will safely and efficiently halt the execution of the computation.

On the other hand, fp-ts does not have built-in support for interruptions. Once a computation starts in fp-ts, it will continue until it completes or encounters an error, without the ability to be interrupted midway.

Defects

The Effect library provides mechanisms for handling defects and managing **unexpected** failures. In Effect, defects refer to unexpected errors or failures that can occur during the execution of a program.

With the Effect library, you have built-in tools and utilities to handle defects in a structured and reliable manner. It offers error handling capabilities that allow you to catch and handle exceptions, recover from failures, and gracefully handle unexpected scenarios.

On the other hand, fp-ts does not have built-in support specifically dedicated to managing defects. While you can handle errors using standard functional programming techniques in fp-ts, the Effect library provides a more comprehensive and streamlined approach to dealing with defects.

Fiber-Based Concurrency

The Effect library leverages fiber-based concurrency, which enables lightweight and efficient concurrent computations. In simpler terms, fiber-based concurrency allows multiple tasks to run concurrently, making your code more responsive and efficient.

With fiber-based concurrency, the Effect library can handle concurrent operations in a way that is lightweight and doesn't block the execution of other tasks. This means that you can run multiple computations simultaneously, taking advantage of the available resources and maximizing performance.

On the other hand, fp-ts does not have built-in support for fiber-based concurrency. While fp-ts provides a rich set of functional programming features, it doesn't have the same level of support for concurrent computations as the Effect library.

Fiber Supervision

Effect library provides supervision strategies for managing and monitoring fibers. fp-ts does not have built-in support for fiber supervision.

Retry and Retry Policies

The Effect library includes built-in support for retrying computations with customizable retry policies. This feature is not available in fp-ts out of the box, and you would need to rely on external libraries to achieve similar functionality. However, it's important to note that the external libraries may not offer the same level of sophistication and fine-tuning as the built-in retry capabilities provided by the Effect library.

Retry functionality allows you to automatically retry a computation or action when it fails, based on a set of predefined rules or policies. This can be particularly useful in scenarios where you are working with unreliable or unpredictable resources, such as network requests or external services.

The Effect library provides a comprehensive set of retry policies that you can customize to fit your specific needs. These policies define the conditions for retrying a computation, such as the number of retries, the delay between retries, and the criteria for determining if a retry should be attempted.

By leveraging the built-in retry functionality in the Effect library, you can handle transient errors or temporary failures in a more robust and resilient manner. This can help improve the overall reliability and stability of your applications, especially in scenarios where you need to interact with external systems or services.

In contrast, fp-ts does not offer built-in support for retrying computations. If you require retry functionality in fp-ts, you would need to rely on external libraries, which may not provide the same level of sophistication and flexibility as the Effect library.

It's worth noting that the built-in retry capabilities of the Effect library are designed to work seamlessly with its other features, such as error handling and resource management. This integration allows for a more cohesive and comprehensive approach to handling failures and retries within your computations.

Built-in Logging

The Effect library comes with built-in logging capabilities. This means that you can easily incorporate logging into your applications without the need for additional libraries or dependencies. In addition, the default logger provided by Effect can be replaced with a custom logger to suit your specific logging requirements.

Logging is an essential aspect of software development as it allows you to record and track important information during the execution of your code. It helps you monitor the behavior of your application, debug issues, and gather insights for analysis.

With the built-in logging capabilities of the Effect library, you can easily log messages, warnings, errors, or any other relevant information at various points in your code. This can be particularly useful for tracking the flow of execution, identifying potential issues, or capturing important events during the operation of your application.

On the other hand, fp-ts does not provide built-in logging capabilities. If you need logging functionality in fp-ts, you would need to rely on external libraries or implement your own logging solution from scratch. This can introduce additional complexity and dependencies into your codebase.

Built-in Scheduling

The Effect library provides built-in scheduling capabilities, which allows you to manage the execution of computations over time. This feature is not available in fp-ts.

In many applications, it's common to have tasks or computations that need to be executed at specific intervals or scheduled for future execution. For example, you might want to perform periodic data updates, trigger notifications, or run background processes at specific times. This is where built-in scheduling comes in handy.

On the other hand, fp-ts does not have built-in scheduling capabilities. If you need to schedule tasks or manage timed computations in fp-ts, you would have to rely on external libraries or implement your own scheduling mechanisms, which can add complexity to your codebase.

Built-in Caching

The Effect library provides built-in caching mechanisms, which enable you to cache the results of computations for improved performance. This feature is not available in fp-ts.

In many applications, computations can be time-consuming or resource-intensive, especially when dealing with complex operations or accessing remote resources. Caching is a technique used to store the results of computations so that they can be retrieved quickly without the need to recompute them every time.

With the built-in caching capabilities of the Effect library, you can easily cache the results of computations and reuse them when needed. This can significantly improve the performance of your application by avoiding redundant computations and reducing the load on external resources.

Built-in Batching

The Effect library offers built-in batching capabilities, which enable you to combine multiple computations into a single batched computation. This feature is not available in fp-ts.

In many scenarios, you may need to perform multiple computations that share similar inputs or dependencies. Performing these computations individually can result in inefficiencies and increased overhead. Batching is a technique that allows you to group these computations together and execute them as a single batch, improving performance and reducing unnecessary processing.

Metrics

The Effect library includes built-in support for collecting and reporting metrics related to computations and system behavior. It specifically supports OpenTelemetry Metrics. This feature is not available in fp-ts.

Metrics play a crucial role in understanding and monitoring the performance and behavior of your applications. They provide valuable insights into various aspects, such as response times, resource utilization, error rates, and more. By collecting and analyzing metrics, you can identify performance bottlenecks, optimize your code, and make informed decisions to improve your application's overall quality.

Tracing

The Effect library has built-in tracing capabilities, which enable you to trace and debug the execution of your code and track the path of a request through an application. Additionally, Effect offers a dedicated OpenTelemetry exporter for integrating with the OpenTelemetry observability framework. In contrast, fp-ts does not offer a similar tracing tool to enhance visibility into code execution.

Configuration

The Effect library provides built-in support for managing and accessing configuration values within your computations. This feature is not available in fp-ts.

Configuration values are an essential aspect of software development. They allow you to customize the behavior of your applications without modifying the code. Examples of configuration values include database connection strings, API endpoints, feature flags, and various settings that

can vary between environments or deployments.

With the Effect library's built-in support for configuration, you can easily manage and access these values within your computations. It provides convenient utilities and abstractions to load, validate, and access configuration values, ensuring that your application has the necessary settings it requires to function correctly.

By leveraging the built-in configuration support in the Effect library, you can:

- Load configuration values from various sources such as environment variables, configuration files, or remote configuration providers.
- Validate and ensure that the loaded configuration values adhere to the expected format and structure.
- Access the configuration values within your computations, allowing you to use them wherever necessary.

Immutable Data Structures

The Effect library provides built-in support for immutable data structures such as `Chunk`, `HashSet`, and `HashMap`. These data structures ensure that once created, their values cannot be modified, promoting safer and more predictable code. In contrast, fp-ts does not have built-in support for such data structures and only provides modules that add additional APIs to standard data types like `Set` and `Map`.

Immutable data structures offer several benefits, including:

- Immutability: Immutable data structures cannot be changed after they are created. This property eliminates the risk of accidental modifications and enables safer concurrent programming.

- Predictability: With immutable data structures, you can rely on the fact that their values won't change unexpectedly. This predictability simplifies reasoning about code behavior and reduces bugs caused by mutable state.

- Sharing and Reusability: Immutable data structures can be safely shared between different parts of your program. Since they cannot be modified, you don't need to create defensive copies, resulting in more efficient memory usage and improved performance.

On the other hand, fp-ts does not have built-in support for these specific immutable data structures. Instead, it provides modules that extend the functionality of standard JavaScript data types like `Set` and `Map` with additional functional programming APIs. While these modules can be useful, they do not offer the same level of performance optimizations and specialized operations as the built-in immutable data structures provided by the Effect library.

Stream Processing

The Effect ecosystem provides built-in support for stream processing, enabling you to work with streams of data. Stream processing is a powerful concept that allows you to efficiently process and transform continuous streams of data in a reactive and asynchronous manner. However, fp-ts does not have this feature built-in and relies on external libraries like RxJS to handle stream processing.

API Reference

Location: 700-other/600-api-reference

Explore the Effect library's API documentation, including core functionalities like effect, CLI, OpenTelemetry, platform, printer, and RPC. Delve into the schema package with getting started and API reference sections. Discover the typeclass module for comprehensive typeclass-related documentation.

- effect
- @effect/cli (Getting Started)
- @effect/opentelemetry
- @effect/platform (Getting Started)
- @effect/printer (Getting Started)
- @effect/rpc
- @effect/schema (Getting Started)
- @effect/typeclass (Getting Started)

Equivalence

Location: 700-other/500-behaviour/equivalence

Equivalence behaviour documentation

This page is a stub. Help us expand it by contributing!

To contribute to the documentation, please join our Discord community at the Docs channel and let us know which part of the documentation you would like to contribute to. We appreciate your help in improving our library's documentation. Thank you!

Higher-Kinded Types

Location: 700-other/500-behaviour/hkt

Explore Higher-Kinded Types (HKTs) and their role in simplifying code and enhancing flexibility. Learn how to create generic structures for different data types, improving code organization and maintainability.

Higher-Kinded Types (HKTs) might sound complex, but they are a valuable concept in programming that can simplify code and make it more flexible. In this article, we'll explore what HKTs are and why they are useful for developers, especially those who are just starting out.

What Are Higher-Kinded Types?

At its core, a higher-kinded type is a type that abstracts over another type, which, in turn, abstracts over yet another type. In simpler terms, it allows us to create generic structures that can work with a wide range of data types. Think of it as a way to build reusable code that can adapt to different data structures.

The Need for HKTs

To understand why HKTs are useful, let's consider a practical scenario. We often want to implement similar functionality across different data structures, like arrays, chunks, and options. Here are some functions as examples:

```
import { Chunk, Option } from "effect"

declare const mapArray: <A, B>(self: Array<A>, f: (a: A) => B) => Array<B>

declare const mapChunk: <A, B>(
  self: Chunk.Chunk<A>,
  f: (a: A) => B
) => Chunk.Chunk<B>

declare const mapOption: <A, B>(
  self: Option.Option<A>,
  f: (a: A) => B
) => Option.Option<B>
```

Notice that these functions share a lot of similarities; in fact, they are almost identical except for the data type they operate on (`Array`, `Chunk`, `Option`).

Now, imagine if we could define a common interface to describe this behavior. This would make our code more organized and easier to maintain. However, doing this in a straightforward way is not so obvious.

The Ideal Solution

In an ideal world, we could create an interface like this:

```
interface Mappable<F<~>> {
  readonly map: <A, B>(self: F<A>, f: (a: A) => B) => F<B>
}
```

With this interface in place, we could do the following:

```
declare const mapArray: Mappable<Array>["map"]
declare const mapChunk: Mappable<Chunk>["map"]
declare const mapOption: Mappable<Option>["map"]
```

We could also define instances of this interface for different data types:

```
declare const ArrayMappable: Mappable<Array>
declare const ChunkMappable: Mappable<Chunk>
declare const OptionMappable: Mappable<Option>
```

Additionally, we could create generic functions like `stringify`:

```
const stringify =
  <F>(T: Mappable<F>) =>
  (self: F<number>): F<string> =>
    T.map(self, (n) => `number: ${n}`)
```

And use them like this:

```
const stringifiedArray: Array<string> = stringify(ArrayMappable)([0, 1, 2])
```

A Brief Terminology

Before we move on, let's clarify some terms:

- `F<~>` is known as a "higher-kinded type".
- The interface `Mappable<F<~>>` is referred to as a "type class".
- Values like `ArrayMappable` are "instances" of the `Mappable` type class.

Now, let's pause our dream scenario and acknowledge that `F<~>` is not valid TypeScript. However, we've grasped the concept of what we'd like to achieve.

In the following sections, we will delve into how HKTs are emulated in Effect. This process involves gradually constructing the essential components needed to work with higher-kinded types effectively.

Type Lambdas

To work effectively with Higher-Kinded Types (HKTs), we need to first grasp the concept of "Type Lambdas." Type Lambdas are a way to define type-level functions in TypeScript, which are not natively supported by the language.

A Type Lambda, like `Target -> F<Target>`, essentially defines a function that operates on types and returns other types. Let's break down this concept:

```
Target -> Array<Target>
```

In this example, the Type Lambda maps the input type `Target` to the output type `Array<Target>`. It's like defining a rule that transforms one type into another.

Type Lambdas allow us to express Higher-Kinded Types directly without the need for complex type definitions.

Implementing a Type Lambda

To implement a Type Lambda, we'll start by defining an interface that includes a `Target` field. Here's how it's done:

```
export interface TypeLambda {
  readonly Target: unknown
}

// @include: TypeLambda
```

This simple interface sets the foundation for our Type Lambdas.

Creating a Type Lambda

Let's create a specific Type Lambda for the `Array` data type:

```
export interface ArrayTypeLambda extends TypeLambda {
  readonly type: Array<this["Target"]>
}

// @include: TypeLambda

// ---cut---
// @include: ArrayTypeLambda
```

Here, we extend the base `TypeLambda` interface to define an `ArrayTypeLambda`. This specific Type Lambda is tailored for working with arrays.

Applying the Type Lambda

Now that we have our Type Lambda and its specialized version for arrays, we need a way to apply this type-level function to a concrete type `A`. We'll call this operator `Kind`:

```
export type Kind<F extends TypeLambda, Target> = F extends {
  readonly type: unknown
}
  ? // If F has a type property, it means it is a concrete type lambda (e.g., F = ArrayTypeLambda).
    // The intersection allows us to obtain the result of applying F to Target.
    (F & {
      readonly Target: Target
    })["type"]
  : // If F is generic, we must explicitly specify all of its type parameters
    // to ensure that none are omitted from type checking.
    {
      readonly F: F
      readonly Target: (_: Target) => Target // This enforces invariance.
    }

// @include: TypeLambda

// ---cut---
// @include: Kind
```

The `Kind` operator takes a Type Lambda `F` and a `Target` type. It ensures that `F` is a valid type lambda and then applies it to the `Target`. This allows us to obtain a type that represents the result of the Type Lambda operation.

Let's test our operator with some examples:

```
// @include: TypeLambda
// @include: Kind
// @include: ArrayTypeLambda

// ---cut---
// Applying ArrayTypeLambda to string
type Test1 = Kind<ArrayTypeLambda, string>
```

```
// Applying ArrayTypeLambda to number
type Test2 = Kind<ArrayTypeLambda, number>
```

Let's take a step further and define Type Lambdas for other data types, such as `Chunk` and `Option`:

```
export interface ChunkTypeLambda extends TypeLambda {
  readonly type: Chunk.Chunk<this["Target"]>
}

export interface OptionTypeLambda extends TypeLambda {
  readonly type: Option.Option<this["Target"]>
}

// @include: TypeLambda
// @include: Kind

// ---cut---
import { Chunk, Option } from "effect"

// @include: ChunkTypeLambda

// @include: OptionTypeLambda

type Test3 = Kind<ChunkTypeLambda, string>

type Test4 = Kind<OptionTypeLambda, string>
```

Type Classes

We are now ready to define the `Mappable` type class, which we introduced earlier:

```
export interface Mappable<F extends TypeLambda> {
  readonly map: <A, B>(self: Kind<F, A>, f: (a: A) => B) => Kind<F, B>
}

// @include: TypeLambda
// @include: Kind

// ---cut---
// @include: Mappable
```

In the code above, we define a `Mappable` type class. This type class provides a blueprint for creating functions that can map values from one type to another. It's a powerful tool for writing code that's both generic and flexible.

Instances

To put our `Mappable` type class to use, we need to create instances for specific data types. These instances will allow us to perform mapping operations on those data types.

```
import { Chunk, Option } from "effect"

export const MappableArray: Mappable<ArrayTypeLambda> = {
  map: (self, f) => self.map(f)
}

export const MappableChunk: Mappable<ChunkTypeLambda> = {
  map: Chunk.map
}

export const MappableOption: Mappable<OptionTypeLambda> = {
  map: Option.map
}

// @include: TypeLambda
// @include: Kind
// @include: ArrayTypeLambda
// @include: ChunkTypeLambda
// @include: OptionTypeLambda
// @include: Mappable

// ---cut---
// @include: instances
```

Here, we've created instances for `Array`, `Chunk`, and `Option` types. Each instance is equipped with a `map` function tailored to its respective data type.

Now, we can proceed to create our `stringify` function:

```
export const stringify =
  <F extends TypeLambda>(TC: Mappable<F>) =>
  (self: Kind<F, number>): Kind<F, string> =>
    TC.map(self, (n) => `number: ${n}`)

// @include: TypeLambda
// @include: Kind
// @include: Mappable
```

```
// ---cut---
// @include: stringify
```

To ensure that everything works as expected, let's run some tests:

```
// @include: TypeLambda
// @include: Kind
// @include: Mappable
// @include: ArrayTypeLambda
// @include: ChunkTypeLambda
// @include: OptionTypeLambda
// @include: instances
// @include: stringify

// ---cut---
const arrayTest = stringify(MappableArray)([1, 2, 3])
console.log(arrayTest)
// [ 'number: 1', 'number: 2', 'number: 3' ]

const chunkTest = stringify(MappableChunk)(Chunk.fromIterable([1, 2, 3]))
console.log(chunkTest)
// { _id: 'Chunk', values: [ 'number: 1', 'number: 2', 'number: 3' ] }

const optionTest = stringify(MappableOption)(Option.some(1))
console.log(optionTest)
// { _id: 'Option', _tag: 'Some', value: 'number: 1' }
```

These tests demonstrate how our `Mappable` type class, `stringify` function, and type instances work together to consistently map values across different data types.

Enhancements

In our current implementation, we've created a simplified version of what Effect provides. However, there is an important enhancement that we need to address. Specifically, we must accommodate more than one parameter, not just `Target`. For instance, certain data types, like `Either<A, E>` or `Effect<A, E, R>`, require two or more type parameters to function correctly.

In Effect, we have the capability to work with data types that can have up to four type parameters, each with distinct variance characteristics. These parameters are essential for defining the behavior and constraints of various data types within Effect. Let's take a closer look at these type parameters:

1. `In` (Contravariant): This type parameter is used for contravariant operations, which means that it accepts input types that are more general or broader than the original type.

2. `Out2` (Covariant): `Out2` represents a covariant type parameter. It allows for operations where the output type is more specific or narrower than the original type.

3. `Out1` (Covariant): Similar to `Out2`, `Out1` is a covariant type parameter, enabling operations that result in a more specific output type.

4. `Target` (Invariant): The `Target` type parameter remains invariant, meaning that it maintains the exact type as the original without any variation.

```
export interface TypeLambda {
  readonly In: unknown
  readonly Out2: unknown
  readonly Out1: unknown
  readonly Target: unknown
}

export type Kind<F extends TypeLambda, In, Out2, Out1, Target> = F extends {
  readonly type: unknown
}
  ? (F & {
      readonly In: In
      readonly Out2: Out2
      readonly Out1: Out1
      readonly Target: Target
    })["type"]
  : {
      readonly F: F
      readonly In: (_: In) => void // Contravariant
      readonly Out2: () => Out2 // Covariant
      readonly Out1: () => Out1 // Covariant
      readonly Target: (_: Target) => Target // Invariant
    }

export declare const URI: unique symbol

export interface TypeClass<F extends TypeLambda> {
  // To improve inference it is necessary to mention the F parameter inside it
  // otherwise it will be lost, we can do so by adding an optional property
  readonly [URI]?: F
}

// @include: HKT
```

Here's how to define a Type Lambda for the `Either` type:

```
// @filename: HKT.ts
// @include: HKT

// @filename: Either.ts
// ---cut---
import { TypeLambda } from "./HKT"
import { Either } from "effect"

export interface EitherTypeLambda extends TypeLambda {
  readonly type: Either.Either<this["Target"], this["Out1"]>
}
```

Please note that we are using the `Out1` parameter, which is covariant since the `E` type parameter of `Either<A, E>` is covariant.

And here's how to define the `Mappable` type class:

```
// @filename: HKT.ts
// @include: HKT

// @filename: Mappable.ts
// ---cut---
import { TypeLambda, TypeClass, Kind } from "./HKT"

export interface Mappable<F extends TypeLambda> extends TypeClass<F> {
  readonly map: <R, O, E, A, B>(
    self: Kind<F, R, O, E, A>,
    f: (a: A) => B
  ) => Kind<F, R, O, E, B>
}
```

Variance

You might be wondering about the purpose of the second branch of the conditional type in the `Kind` type.

This second branch serves to enforce something called "variance." To understand this concept, let's explore an example. Imagine we define a type class like this:

```
import { Kind, TypeClass, TypeLambda } from "./HKT"

export interface Zippable<F extends TypeLambda> extends TypeClass<F> {
  readonly zip: <R1, O1, E1, A, R2, O2, E2, B>(
    first: Kind<F, R1, O1, E1, A>,
    second: Kind<F, R2, O2, E2, B>
  ) => Kind<F, R1 & R2, O1 | O2, E1 | E2, readonly [A, B]>
}
```

```
// @filename: HKT.ts
// @include: HKT

// @filename: Zippable.ts
// ---cut---
// @include: Zippable
```

Now, we derive a `pipe`-able version of `zip`:

```
// @filename: HKT.ts
// @include: HKT

// @filename: Zippable.ts
// @include: Zippable
// ---cut---
export const zip =
  <F extends TypeLambda>(Zippable: Zippable<F>) =>
  <R2, O2, E2, B>(that: Kind<F, R2, O2, E2, B>) =>
  <R1, O1, E1, A>(
    self: Kind<F, R1, O1, E1, A>
  ): Kind<F, R1 & R2, O1 | O2, E1 | E2, readonly [A, B]> =>
    Zippable.zip(self, that)
```

However, let's assume that we make a mistake while typing the return type of `zip` by specifying `R1` instead of `R1 & R2`:

```
-  ): Kind<F, R1 & R2, O1 | O2, E1 | E2, readonly [A, B]> =>
+  ): Kind<F, R1, O1 | O2, E1 | E2, readonly [A, B]> =>
```

In this case, it will not type check, and you'll encounter the following error:

```
...
Types of property 'In' are incompatible.
  Type '(_: R1 & R2) => void' is not assignable to type '(_: R1) => void'.
    Types of parameters '_' and '_' are incompatible.
      Type 'R1' is not assignable to type 'R1 & R2'.ts(2322)
```

The second branch of the conditional type helps catch such type errors and ensures that the type parameters are correctly aligned, enforcing proper type checking.

Now, let's proceed to define an instance of `Zippable` for the `Either` type:

```
// @filename: HKT.ts
// @include: HKT

// @filename: Zippable.ts
// @include: Zippable

// @filename: Either.ts
// ---cut---
import { TypeLambda } from "./HKT"
import { Either } from "effect"
import { Zippable } from "./Zippable"

export interface EitherTypeLambda extends TypeLambda {
  readonly type: Either.Either<this["Target"], this["Out1"]>
}

export const EitherZippable: Zippable<EitherTypeLambda> = {
  zip: (first, second) => {
    if (Either.isLeft(first)) {
      return Either.left(first.left)
    }
    if (Either.isLeft(second)) {
      return Either.left(second.left)
    }
    return Either.right([first.right, second.right])
  }
}
```

If you hover over `EitherZippable.zip` you will notice that the return type is as follows:

```
Either<readonly [A, B], E1 | E2>
```

This signifies that the system has correctly managed the covariance of the `E` type parameter by returning the union of possible errors: `E1 | E2`.

Order

Location: 700-other/500-behaviour/order

Explore the Order module in Effect, which provides a powerful interface for comparing and sorting values. Learn about built-in comparators for common data types, sorting arrays, deriving custom orders, combining orders, and additional useful functions for comparisons, finding minimum/maximum, clamping values, and checking value range.

The Order module provides a way to compare values and determine their order. It defines an interface `Order<A>` which represents a single function for comparing two values of type `A`. The function returns `-1`, `0`, or `1`, indicating whether the first value is less than, equal to, or greater than the second value.

Here's the basic structure of an `Order`:

```
interface Order<A> {
  (first: A, second: A): -1 | 0 | 1
}
```

Using the Built-in Orders

The Order module comes with several built-in comparators for common data types:

- `string`: Used for comparing strings.
- `number`: Used for comparing numbers.
- `bigint`: Used for comparing big integers.
- `Date`: Used for comparing `Date`s.

Here's how you can use these comparators:

```
// @target: ES2020
import { Order } from "effect"

console.log(Order.string("apple", "banana"))
// Output: -1, as "apple" < "banana"
console.log(Order.number(1, 1))
// Output: 0, as 1 = 1
console.log(Order.bigint(2n, 1n))
// Output: 1, as 2n > 1n
```

Sorting Arrays

Once you have your comparators, you can sort arrays. The Array module provides a handy function called `sort` that allows you to sort arrays without modifying the original array. Here's an example:

```
import { Order, Array } from "effect"

const strings = ["b", "a", "d", "c"]

const result = Array.sort(strings, Order.string)

console.log(strings)
console.log(result)
/*
Output:
[ 'b', 'a', 'd', 'c' ]
[ 'a', 'b', 'c', 'd' ]
*/
```

You can even use an `Order` as a comparator in JavaScript's native `Array.sort` method:

```
import { Order } from "effect"

const strings = ["b", "a", "d", "c"]

strings.sort(Order.string)

console.log(strings)
// Output: [ 'a', 'b', 'c', 'd' ]
```

Please note that when using `Array#sort`, you modify the original array. So, be cautious if you want to keep the original order. If you don't want to alter the original array, consider using `Array.sort` as shown earlier.

Deriving Orders

Sometimes, when working with more complex data structures, you may need to create custom sorting rules. The `Order` module allows you to do this by deriving a new `Order` from an existing one using the `Order.mapInput` function.

Imagine you have a list of `Person` objects, and you want to sort them by their names in ascending order. To achieve this, you can create a custom `Order`.

Here's how you can do it:

```
import { Order, Array } from "effect"

// Define the Person interface
interface Person {
  readonly name: string
  readonly age: number
}

// Create a custom sorting rule to sort Persons by their names in ascending order
const byName = Order.mapInput(Order.string, (person: Person) => person.name)
```

In this example, we first import the necessary modules and define the `Person` interface, representing our data structure. Next, we create a custom sorting rule called `byName` using the `mapInput` function.

The `mapInput` function takes two arguments:

1. The existing sorting rule you want to use as a base (`Order.string` in this case, for comparing strings).
2. A function that extracts the value you want to use for sorting from your data structure (`person.name` in this case).

Once you have defined your custom sorting rule, you can apply it to sort a list of `Person` objects:

```
import { Order, Array } from "effect"

interface Person {
  readonly name: string
  readonly age: number
}

const byName = Order.mapInput(Order.string, (person: Person) => person.name)

// ---cut---
const persons: ReadonlyArray<Person> = [
  { name: "Charlie", age: 22 },
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 }
]

// Use your custom sorting rule to sort the persons array
const sortedPersons = Array.sort(persons, byName)

console.log(sortedPersons)
/*
Output:
[
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 22 }
```

```
]
*/
```

Combining Orders

The Order module not only handles basic comparisons but also empowers you to create intricate sorting rules. This is especially valuable when you need to sort data based on multiple criteria or properties.

The `combine*` functions in the Order module enables you to merge two or more `Order` instances, resulting in a new `Order` that incorporates the combined sorting logic. Let's walk through an example to illustrate this concept.

Imagine you have a list of people, each represented by an object with a `name` and an `age`. You want to sort this list first by name and then, for individuals with the same name, by age.

Here's how you can achieve this using the Order module:

```
import { Order, Array } from "effect"

// Define the structure of a person
interface Person {
  readonly name: string
  readonly age: number
}

// Create an Order to sort people by their names
const byName = Order.mapInput(Order.string, (person: Person) => person.name)

// Create an Order to sort people by their ages
const byAge = Order.mapInput(Order.number, (person: Person) => person.age)

// Combine the two Orders to create a complex sorting logic
const byNameAge = Order.combine(byName, byAge)

const result = Array.sort(
  [
    { name: "Bob", age: 20 },
    { name: "Alice", age: 18 },
    { name: "Bob", age: 18 }
  ],
  byNameAge
)

console.log(result)
/*
Output:
[
  { name: 'Alice', age: 18 }, <-- by name
  { name: 'Bob', age: 18 },   <-- by age
  { name: 'Bob', age: 20 }    <-- by age
]
*/
```

In the code above, we first create two separate `Order` instances: `byName` and `byAge`. These orders individually sort people by their names and ages, respectively.

Next, we use the `combine` function to merge these two orders into a single `byNameAge` order. This combined order first sorts people by name and then, for those with the same name, by age.

Finally, we apply this combined order to the array of people using `Array.sort`. The result is an array of people sorted according to the specified criteria.

Additional Useful Functions

The Order module extends its functionality by providing additional functions for common operations. These functions make it easier to work with ordered values and perform various comparisons. Let's explore each of them:

Reversing Order

The `Order.reverse` function does exactly what its name suggests; it reverses the order of comparison. If you have an `Order` that sorts values in ascending order, applying `reverse` will sort them in descending order.

```
import { Order } from "effect"

const ascendingOrder = Order.number
const descendingOrder = Order.reverse(ascendingOrder)

console.log(ascendingOrder(1, 3))
// Output: -1 (1 < 3 in ascending order)
console.log(descendingOrder(1, 3))
// Output: 1 (1 > 3 in descending order)
```

Comparing Values

These functions allow you to perform simple comparisons between values:

- `lessThan`: Checks if one value is strictly less than another.
- `greaterThan`: Checks if one value is strictly greater than another.
- `lessThanOrEqualTo`: Checks if one value is less than or equal to another.
- `greaterThanOrEqualTo`: Checks if one value is greater than or equal to another.

```
import { Order } from "effect"

console.log(Order.lessThan(Order.number)(1, 2))
// Output: true (1 < 2)
console.log(Order.greaterThan(Order.number)(5, 3))
// Output: true (5 > 3)
console.log(Order.lessThanOrEqualTo(Order.number)(2, 2))
// Output: true (2 <= 2)
console.log(Order.greaterThanOrEqualTo(Order.number)(4, 4))
// Output: true (4 >= 4)
```

Finding Minimum and Maximum

The `min` and `max` functions return the minimum or maximum value between two values, considering the order. These functions are particularly useful when you want to determine the smallest or largest value among multiple options.

```
import { Order } from "effect"

console.log(Order.min(Order.number)(3, 1))
// Output: 1 (1 is the minimum)
console.log(Order.max(Order.number)(5, 8))
// Output: 8 (8 is the maximum)
```

Clamping Values

The `clamp` function ensures that a value stays within a specified range. It takes three arguments: the value you want to clamp, the minimum bound, and the maximum bound. If the value falls outside the range, it's adjusted to the nearest bound.

```
import { Order } from "effect"

const clampedValue = Order.clamp(Order.number)(10, {
  minimum: 20,
  maximum: 30
})

console.log(clampedValue)
// Output: 20 (10 is clamped to the nearest bound, which is 20)
```

Checking Value Range

The `between` function checks if a value falls within a specified range, inclusively. It takes three arguments: the value you want to check, the minimum bound, and the maximum bound.

```
import { Order } from "effect"

console.log(Order.between(Order.number)(15, { minimum: 10, maximum: 20 }))
// Output: true (15 is within the range [10, 20])
console.log(Order.between(Order.number)(5, { minimum: 10, maximum: 20 }))
// Output: false (5 is outside the range [10, 20])
```

Behaviours

Location: 700-other/500-behaviour/index

Behaviours

What is Dependency Injection?

Location: 600-concepts/200-dependency-injection

Understand the Dependency Injection design pattern in software development, fostering loose coupling by passing dependencies externally. Learn its benefits, illustration of problems, and solutions through decoupling the dependency graph and using service interfaces.

Dependency injection is a design pattern used in software development that helps manage the dependencies between different components of an application. It allows developers to create loosely coupled code by passing dependencies to a class or function from an external source.

In simpler terms, instead of creating dependencies inside a component, dependency injection enables us to provide them from the outside, making code more flexible and easier to test and maintain. By using dependency injection, developers can easily swap out dependencies or change their behavior without modifying the component's code directly.

Illustrating the Problem

Let's assume we have a `Mailer` service which depends upon the functionality provided by a `Logger` service.

```
export class Logger {
  log(message: string): void {
    console.log(message)
  }
}

export class Mailer {
  logger = new Logger()
  sendMail(address: string, message: string): void {
    this.logger.log(`Sending the message ${message} to ${address}`)
  }
}
```

In the code above, we directly construct a `Logger` within our `Mailer` service. This tight coupling between the `Logger` and `Mailer` services introduces several problems:

1. Developers using the `Mailer` service have no control over how the `Logger` is constructed
2. Alternate implementations of the `Logger` service (e.g. for testing) cannot be provided
3. Changes to the `Logger` service may necessitate changes to the `Mailer` service

However, by making use of dependency injection we can decouple our `Mailer` and `Logger` services and solve the problems outlined above.

Decoupling the Dependency Graph

The first step to solving the problems outlined above is to decouple the construction of the `Mailer` and `Logger` services from one another.

```
export class Logger {
  log(message: string): void {
    console.log(message)
  }
}

export class Mailer {
  constructor(readonly logger: Logger) {}
  sendMail(address: string, message: string): void {
    this.logger.log(`Sending the message ${message} to ${address}`)
  }
}

const logger = new Logger()
const mailer = new Mailer(logger)
```

Now instead of constructing the `Logger` service within the `Mailer` service, we construct the `Logger` externally and pass it to the constructor of the `Mailer` service.

<Idea> This pattern of inverting control of a service to the user is commonly known in software engineering as *Inversion of Control*. </Idea>

This gives the developer much more control over how the dependencies within their application are constructed and composed together into a dependency graph.

Using Service Interfaces

Though we have improved the coupling between our `Mailer` and `Logger` services in the example above, there is still a problem with our code - we can only have a single implementation of our `Mailer` and `Logger` services.

But what if we want to provide a different implementation of `Logger` to the `Mailer` service when we are running our application's test suite?

We can take things a step further and allow for multiple implementations of our services by decoupling the *interface* of our services from the *implementation* of the service.

<Steps>

Define the interface of our services

First, we define the *interface*, or *behavior*, that our services should expose:

```
export interface Logger {
  log(message: string): void
}

export interface Mailer {
  sendMail(address: string, message: string): void
}
```

Create concrete service implementations

Then, we bind the actual implementation of these services to the interfaces we have defined:

```
export interface Logger {
  log(message: string): void
}

export interface Mailer {
  sendMail(address: string, message: string): void
}

// ---cut---
class ConsoleLogger implements Logger {
  log(message: string): void {
    console.log(message)
  }
}

class ConsoleMailer implements Mailer {
  constructor(readonly logger: Logger) {}
  sendMail(address: string, message: string): void {
    this.logger.log(`Sending the message ${message} to ${address}`)
  }
}

const logger = new ConsoleLogger()
const mailer = new ConsoleMailer(logger)
```

Providing other service implementations

Now, as long as we adhere to the interface specified by our services, we can easily create alternate implementations.

For example, we can create mock implementations of the `Logger` and `Mailer` services to use in our tests which internally track all messages logged and sent by the services:

```
export interface Logger {
  log(message: string): void
}

export interface Mailer {
  sendMail(address: string, message: string): void
}

// ---cut---
class MockLogger implements Logger {
  readonly messages: Array<string> = []
  log(message: string): void {
    this.messages.push(message)
  }
}

class MockMailer implements Mailer {
  readonly sentMail: Array<string> = []
  constructor(readonly logger: Logger) {}
  sendMail(address: string, message: string): void {
    const email = `Sending the message ${message} to ${address}`
    this.logger.log(email)
    this.sentMail.push(email)
  }
}

const mockLogger = new MockLogger()
const mockMailer = new MockMailer(mockLogger)
```

</Steps>

Concepts

Location: 600-concepts/index

Concepts

Immutability

Location: 600-concepts/100-immutability

Explore the importance and benefits of immutability in programming, providing enhanced code reliability and maintainability. Learn key concepts and practices to incorporate immutability into your projects.

<Stub />

Integrations

Location: 500-integrations/index

Integrations

Express Integration

Location: 500-integrations/100-express

Explore integrating Effect with Express, a popular Node.js web framework. Learn to create a simple Express server with "Hello World!" response and understand basic routing with Effect and Express. Follow the guide to set up, run, and breakdown the provided examples.

In this guide, we'll explore how to integrate Effect with Express, a popular web framework for Node.js.

Hello World Example

Let's start with a simple example that creates an Express server responding with "Hello World!" for requests to the root URL (/). This mirrors the classic "Hello world example" found in the Express documentation.

Setup Steps

1. Create a new directory for your project and navigate to it using your terminal:

   ```
   mkdir express-effect-integration
   cd express-effect-integration
   ```

2. Initialize your project with npm. This will create a `package.json` file:

   ```
   npm init -y
   ```

3. Install the necessary dependencies:

   ```
   npm install effect express
   ```

   Install the necessary dev dependencies:

   ```
   npm install typescript @types/express --save-dev
   ```

   Now, initialize TypeScript:

   ```
   npx tsc --init
   ```

4. Create a new file, for example, `hello-world.ts`, and add the following code:

   ```
   import { Context, Layer, Effect, Runtime } from "effect"
   import express from "express"

   // Define Express as a service
   class Express extends Context.Tag("Express")<
     Express,
     ReturnType<typeof express>
   >() {}

   // Define the main route, IndexRouteLive, as a Layer
   const IndexRouteLive = Layer.effectDiscard(
     Effect.gen(function* () {
       const app = yield* Express
       const runFork = Runtime.runFork(yield* Effect.runtime<never>())

       app.get("/", (_, res) => {
         runFork(Effect.sync(() => res.send("Hello World!")))
       })
     })
   )

   // Server Setup
   const ServerLive = Layer.scopedDiscard(
     Effect.gen(function* () {
       const port = 3001
       const app = yield* Express
       yield* Effect.acquireRelease(
         Effect.sync(() =>
           app.listen(port, () =>
             console.log(`Example app listening on port ${port}`)
           )
         ),
         (server) => Effect.sync(() => server.close())
       )
     })
   )

   // Setting Up Express
   const ExpressLive = Layer.sync(Express, () => express())
   ```

```
// Combine the layers
const AppLive = ServerLive.pipe(
  Layer.provide(IndexRouteLive),
  Layer.provide(ExpressLive)
)

// Run the program
Effect.runFork(Layer.launch(AppLive))
```

5. Run your Express server. Here we are using ts-node to run the `hello-world.ts` file in the terminal:

   ```
   npx ts-node hello-world.ts
   ```

   Visit http://localhost:3001 in your web browser, and you should see "Hello World!".

Code Breakdown

Here's a breakdown of what's happening:

- **Express Service**. We define an `Express` service to retrieve the Express app later on.

  ```
  // Define Express as a service
  class Express extends Context.Tag("Express")<
    Express,
    ReturnType<typeof express>
  >() {}
  ```

- **Main Route**. The main route, `IndexRouteLive`, is defined as a Layer.

  ```
  // Define the main route, IndexRouteLive, as a Layer
  const IndexRouteLive = Layer.effectDiscard(
    Effect.gen(function* () {
      const app = yield* Express
      const runFork = Runtime.runFork(yield* Effect.runtime<never>())

      app.get("/", (_, res) => {
        runFork(Effect.sync(() => res.send("Hello World!")))
      })
    })
  )
  ```

  We access the runtime (`Effect.runtime`), which can be used to execute tasks within our route (`runFork`). Since we don't need to produce any service in the output, we use `Layer.effectDiscard` to discard its output.

- **Server Setup**. The server is created in a layer (`ServerLive`) and mounted at the end of our program.

  ```
  // Server Setup
  const ServerLive = Layer.scopedDiscard(
    Effect.gen(function* () {
      const port = 3001
      const app = yield* Express
      yield* Effect.acquireRelease(
        Effect.sync(() =>
          app.listen(port, () =>
            console.log(`Example app listening on port ${port}`)
          )
        ),
        (server) => Effect.sync(() => server.close())
      )
    })
  )
  ```

  We use Effect.acquireRelease to create the server, allowing automatic management of the scope. Again, as we don't need to produce any service in the output, we use `Layer.scopedDiscard` to discard its output.

- **Mounting**. Finally, we mount the server by adding our route

  ```
  const AppLive = ServerLive.pipe(
    Layer.provide(IndexRouteLive),
    Layer.provide(ExpressLive)
  )
  ```

  and providing the necessary dependency to the Express app

  ```
  const ExpressLive = Layer.sync(Express, () => express())

  // Combine the layers
  const AppLive = ServerLive.pipe(
    Layer.provide(IndexRouteLive),
    Layer.provide(ExpressLive)
  )
  ```

Basic routing

In this example, we'll explore the basics of routing with Effect and Express. The goal is to create a simple web server with two routes: one that returns all todos and another that returns a todo by its ID.

```
import { Context, Effect, FiberSet, Layer } from "effect"
import express from "express"

//
// Express
//
// NB: this is an example of an integration to a third party lib, not the suggested way of integrating express
//

// Define Express as a service
class Express extends Context.Tag("Express")<
  Express,
  ReturnType<typeof express>
>() {}

const get = <A, E, R>(
  path: string,
  body: (
    req: express.Request,
    res: express.Response
  ) => Effect.Effect<A, E, R>
) =>
  Effect.gen(function* () {
    const app = yield* Express
    const run = yield* FiberSet.makeRuntime<R>()
    app.get(path, (req, res) => run(body(req, res)))
  })

// Server Setup
const ServerLive = Layer.scopedDiscard(
  Effect.gen(function* () {
    const port = 3001
    const app = yield* Express
    yield* Effect.acquireRelease(
      Effect.sync(() =>
        app.listen(port, () =>
          console.log(`Example app listening on port ${port}`)
        )
      ),
      (server) => Effect.sync(() => server.close())
    )
  })
)

// Setting Up Express
const ExpressLive = Layer.sync(Express, () => express())

//
// Domain
//

interface Todo {
  readonly id: number
  readonly title: string
  readonly completed: boolean
}

// Define the repository as a service
class TodoRepository extends Context.Tag("TodoRepository")<
  TodoRepository,
  {
    readonly getTodos: Effect.Effect<Array<Todo>>
    readonly getTodo: (id: number) => Effect.Effect<Todo | null>
  }
>() {}

//
// App
//

// Define a main route that returns all Todos
const IndexRouteLive = Layer.scopedDiscard(
  Effect.gen(function* () {
    const repo = yield* TodoRepository

    yield* get("/", (_, res) =>
      Effect.gen(function* () {
        const todos = yield* repo.getTodos
        res.json(todos)
      })
    )
  })
)

// Define a route that returns a Todo by its ID
const TodoByIdRouteLive = Layer.scopedDiscard(
  Effect.gen(function* () {
```

```
    const repo = yield* TodoRepository

    yield* get("/todo/:id", (req, res) =>
      Effect.gen(function* () {
        const id = req.params.id
        const todo = yield* repo.getTodo(Number(id))
        res.json(todo)
      })
    )
  })
)

// Merge routes into a single layer
const RouterLive = Layer.mergeAll(IndexRouteLive, TodoByIdRouteLive)

// Combine all layers to create the final application layer
const AppLive = ServerLive.pipe(
  Layer.provide(RouterLive),
  Layer.provide(ExpressLive)
)

// Test Data for TodoRepository
const testData = [
  {
    id: 1,
    title: "delectus aut autem",
    completed: false
  },
  {
    id: 2,
    title: "quis ut nam facilis et officia qui",
    completed: false
  },
  {
    id: 3,
    title: "fugiat veniam minus",
    completed: false
  }
]

// Create a layer with test data
const TodoRepositoryTest = Layer.succeed(TodoRepository, {
  getTodos: Effect.succeed(testData),
  getTodo: (id) =>
    Effect.succeed(testData.find((todo) => todo.id === id) || null)
})

const Test = AppLive.pipe(Layer.provide(TodoRepositoryTest))

Effect.runFork(Layer.launch(Test))
```

Quickstart

Location: 300-quickstart

Learn how to set up a new Effect project from scratch in TypeScript, covering Node.js, Deno, Bun, and Vite + React environments. Follow step-by-step instructions for each platform to create a basic program using the Effect library.

In this tutorial, we will guide you through the process of setting up a new Effect project from scratch using plain **TypeScript 5.4 or newer**.

Why Effect?

Location: 200-why-effect

Effect presents a new way of thinking about programming in TypeScript, offering an ecosystem of tools to build better applications and libraries. Learn how to leverage the TypeScript type system to enhance reliability and maintainability.

Motivation

Programming is challenging. When we build libraries and apps, we look to many tools to handle the complexity and make our day-to-day more manageable. Effect presents a new way of thinking about programming in TypeScript.

Effect is an ecosystem of tools that help you build better applications and libraries. As a result, you will also learn more about the TypeScript language and how to use the type system to make your programs more reliable and easier to maintain.

In "typical" TypeScript, without Effect, we write code that assumes that a function is either successful or throws an exception. Take this trivial example of division:

```
const divide = (a: number, b: number): number => {
  if (b === 0) {
    throw new Error("Cannot divide by zero")
  }
```

```
  return a / b
}
```

Based on the types, we have no idea that this function can throw an exception. We can only find out by reading the code. This may not seem like much of a problem when you only have one function in your codebase, but when you have hundreds or thousands, it really starts to add up. It's easy to forget that a function can throw an exception, and it's easy to forget to handle that exception.

Often, we will do the "easiest" thing and just wrap the function in a `try/catch` block. This is a good first step to prevent your program from crashing, but it doesn't make it any easier to manage or understand our complex application/library. We can do better.

One of the most important tools we have in TypeScript is the compiler. It is the first line of defense against bugs, domain errors, and general complexity.

The Effect Pattern

While Effect is a vast ecosystem of many different tools, if it had to be reduced down to just one idea, it would be the following:

Effect's major unique insight is that we can use the type system to track *errors* and *"context"* (more on this later), not only *success* values as shown in the divide example above.

Here's the same divide function from above, but with the Effect pattern:

```
import { Effect } from "effect"

const divide = (a: number, b: number): Effect.Effect<number, Error, never> =>
  b === 0
    ? Effect.fail(new Error("Cannot divide by zero"))
    : Effect.succeed(a / b)
```

Notice how looking at the type signature tells you exactly what success value(s) it returns (`number`), what error(s) it can throw (`Error`) and what context the function needs (`never`, as in, no context required here). This function no longer throws an exception, and you can cleanly pass on the error to the caller of this function.

Errors now become values, just like your success values. Effect gives us many functions to make managing errors and success values ergonomic.

Additionally, tracking context allows you to provide additional information to your functions without having to pass in everything as an argument. For example, you can swap out implementations of live external services with mocks during your tests without changing any core business logic. There are many other use cases for context as well.

Effect's Ecosystem

It turns out that this unique insight, along with a lot of other tooling, has led to a rich ecosystem of libraries that make building complex applications in TypeScript a breeze. Things that used to seem impossible are now ordinary. Effect's ecosystem is growing rapidly, and you can find the growing list on Effect's GitHub.

Don't Re-Invent the Wheel

Application code in TypeScript often solves the same problems over and over again. Interacting with external services, filesystems, databases, etc. are common problems for all application developers. Effect provides a rich ecosystem of libraries that provide standardized solutions to many of these problems. You can use these libraries to build your application, or you can use them to build your own libraries.

Managing challenges like error handling, debugging, tracing, async/promises, retries, streaming, concurrency, caching, resource management, and a lot more are made manageable with Effect. You don't have to re-invent the solutions to these problems, or install tons of dependencies. Effect, under one umbrella, solves many of the problems that you would usually install many different dependencies with different APIs to solve.

Solving Practical Problems

Effect is heavily inspired by great work done in other languages, like Scala and Haskell. However, it's important to understand that Effect's goal is to be a practical toolkit, and it goes to great lengths to solve real, everyday problems that developers face when building applications and libraries in TypeScript.

Enjoy Building and Learning

Learning Effect is a lot of fun. Many developers in the Effect ecosystem are using Effect to solve real problems in their day-to-day work, and also experiment with cutting edge ideas for pushing TypeScript to be the most useful language it can be.

You don't have to use all aspects of Effect at once, and can start with the pieces of the ecosystem that make the most sense for the problems you are solving. Effect is a toolkit, and you can pick and choose the pieces that make the most sense for your use case. However, as more and more of your codebase is using Effect, you will probably find yourself wanting to utilize more of the ecosystem!

Effect's concepts may be new to you, and might not completely make sense at first. This is totally normal. Take your time with reading the docs and try to understand the core concepts - this will really pay off later on as you get into the more advanced tooling in the Effect ecosystem. The Effect community is always happy to help you learn and grow. Feel free to hop into our Discord or discuss on GitHub! We are open to feedback and contributions, and are always looking for ways to improve Effect.

Welcome to Effect

Location: 100-introduction

Effect is a powerful TypeScript library designed to help developers easily create complex, synchronous, and asynchronous programs.

Welcome to the Effect documentation!

What is Effect?

Effect is a powerful TypeScript library designed to help developers easily create complex, synchronous, and asynchronous programs.

Main Features

Some of the main Effect features include:

Feature	Description
Concurrency	Achieve highly-scalable, ultra low-latency applications through Effect's fiber-based concurrency model.
Composability	Construct highly maintainable, readable, and flexible software through the use of small, reusable building blocks.
Resource Safety	Safely manage acquisition and release of resources, even when your program fails.
Type Safety	Leverage the TypeScript type system to the fullest with Effect's focus on type inference and type safety.
Error Handling	Handle errors in a structured and reliable manner using Effect's built-in error handling capabilities.
Asynchronicity	Write code that looks the same, whether it is synchronous or asynchronous.
Observability	With full tracing capabilities, you can easily debug and monitor the execution of your Effect program.

How to Use These Docs

The documentation is structured in a sequential manner, starting from the basics and progressing to more advanced topics. This allows you to follow along step-by-step as you build your Effect application. However, you have the flexibility to read the documentation in any order or jump directly to the pages that are relevant to your specific use case.

To facilitate navigation within a page, you will find a table of contents on the right side of the screen. This allows you to easily jump between different sections of the page.

Join our Community

If you have questions about anything related to Effect, you're always welcome to ask our community on Discord.

Branded Types

Location: 400-guides/720-style/200-branded-types

Explore the concept of branded types in TypeScript using the Brand module. Understand the "data-last" and "data-first" variants, and learn to create branded types with runtime validation (refined) and without checks (nominal). Discover how to use and combine branded types to enforce type safety in your code.

In this guide, we will explore the concept of **branded types** in TypeScript and learn how to create and work with them using the Brand module. Branded types are TypeScript types with an added type tag that helps prevent accidental usage of a value in the wrong context. They allow us to create distinct types based on an existing underlying type, enabling type safety and better code organization.

The Problem with TypeScript's Structural Typing

TypeScript's type system is structurally typed, meaning that two types are considered compatible if their members are compatible. This can lead to situations where values of the same underlying type are used interchangeably, even when they represent different concepts or have different meanings.

Consider the following types:

```
type UserId = number

type ProductId = number
```

Here, `UserId` and `ProductId` are structurally identical as they are both based on `number`. TypeScript will treat these as interchangeable, potentially causing bugs if they are mixed up in your application.

For example:

```
type UserId = number

type ProductId = number
```

```
const getUserById = (id: UserId) => {
  // Logic to retrieve user
}

const getProductById = (id: ProductId) => {
  // Logic to retrieve product
}

const id: UserId = 1

getProductById(id) // No type error, but this is incorrect usage
```

In the example above, passing a `UserId` to `getProductById` should ideally throw a type error, but it doesn't due to structural compatibility.

How Branded Types Help

Branded types allow you to create distinct types from the same underlying type by adding a unique type tag, enforcing proper usage at compile-time.

Branding is accomplished by adding a symbolic identifier that distinguishes one type from another at the type level. This method ensures that types remain distinct without altering their runtime characteristics.

Let's start by introducing the `BrandTypeId` symbol:

```
const BrandTypeId: unique symbol = Symbol.for("effect/Brand")

type ProductId = number & {
  readonly [BrandTypeId]: {
    readonly ProductId: "ProductId" // unique identifier for ProductId
  }
}
```

This approach assigns a unique identifier as a brand to the `number` type, effectively differentiating `ProductId` from other numerical types. The use of a symbol ensures that the branding field does not conflict with any existing properties of the `number` type.

Attempting to use a `UserId` in place of a `ProductId` now results in an error:

```
// @errors: 2345
const BrandTypeId: unique symbol = Symbol.for("effect/Brand")

type ProductId = number & {
  readonly [BrandTypeId]: {
    readonly ProductId: "ProductId"
  }
}
// ---cut---
const getProductById = (id: ProductId) => {
  // Logic to retrieve product
}

type UserId = number

const id: UserId = 1

getProductById(id)
```

The error message clearly states that a `number` cannot be used in place of a `ProductId`.

TypeScript won't let us pass an instance of `number` to the function accepting `ProductId` because it's missing the brand field.

What if `UserId` also had its own brand?

```
// @errors: 2345
const BrandTypeId: unique symbol = Symbol.for("effect/Brand")

type ProductId = number & {
  readonly [BrandTypeId]: {
    readonly ProductId: "ProductId" // unique identifier for ProductId
  }
}

const getProductById = (id: ProductId) => {
  // Logic to retrieve product
}

type UserId = number & {
  readonly [BrandTypeId]: {
    readonly UserId: "UserId" // unique identifier for UserId
  }
}

declare const id: UserId

getProductById(id)
```

The error is saying that though both types utilize a branding strategy, the distinct values associated with their branding fields (`"ProductId"` and `"UserId"`) prevent them from being interchangeable.

Generalizing Branded Types

To enhance the versatility and reusability of branded types, they can be generalized using a standardized approach:

```
const BrandTypeId: unique symbol = Symbol.for("effect/Brand")

// Create a generic Brand interface using a unique identifier
interface Brand<in out ID extends string | symbol> {
  readonly [BrandTypeId]: {
    readonly [id in ID]: ID
  }
}

// Define a ProductId type branded with a unique identifier
type ProductId = number & Brand<"ProductId">

// Define a UserId type branded similarly
type UserId = number & Brand<"UserId">
```

This design allows any type to be branded using a unique identifier, either a string or symbol.

Here's how you can utilize the `Brand` interface, which is readily available from the Brand module, eliminating the need to craft your own implementation:

```
import { Brand } from "effect"

// Define a ProductId type branded with a unique identifier
type ProductId = number & Brand.Brand<"ProductId">

// Define a UserId type branded similarly
type UserId = number & Brand.Brand<"UserId">
```

However, creating instances of these types directly leads to an error because the type system expects the brand structure:

```
// @errors: 2322
const BrandTypeId: unique symbol = Symbol.for("effect/Brand")

interface Brand<in out K extends string | symbol> {
  readonly [BrandTypeId]: {
    readonly [k in K]: K
  }
}

type ProductId = number & Brand<"ProductId">
// ---cut---
const id: ProductId = 1
```

We need a way to create a value of type `ProductId` without directly assigning a number to it. This is where the Brand module comes in.

Constructing Branded Types

The Brand module offers two core functions for constructing branded types: `nominal` and `refined`.

nominal

The `nominal` function is designed for defining branded types that do not require runtime validations. It simply adds a type tag to the underlying type, allowing us to distinguish between values of the same type but with different meanings. Nominal branded types are useful when we only want to create distinct types for clarity and code organization purposes.

```
import { Brand } from "effect"

type UserId = number & Brand.Brand<"UserId">

// Constructor for UserId
const UserId = Brand.nominal<UserId>()

const getUserById = (id: UserId) => {
  // Logic to retrieve user
}

type ProductId = number & Brand.Brand<"ProductId">

// Constructor for ProductId
const ProductId = Brand.nominal<ProductId>()

const getProductById = (id: ProductId) => {
  // Logic to retrieve product
}
```

Attempting to assign a non-`ProductId` value will result in a compile-time error:

```
// @errors: 2345
import { Brand } from "effect"

type UserId = number & Brand.Brand<"UserId">

const UserId = Brand.nominal<UserId>()

const getUserById = (id: UserId) => {
  // Logic to retrieve user
}

type ProductId = number & Brand.Brand<"ProductId">

const ProductId = Brand.nominal<ProductId>()

const getProductById = (id: ProductId) => {
  // Logic to retrieve product
}
// ---cut---
// Correct usage
getProductById(ProductId(1))

// Incorrect, will result in an error
getProductById(1)

// Also incorrect, will result in an error
getProductById(UserId(1))
```

refined

The `refined` function enables the creation of branded types that include data validation. It requires a refinement predicate to check the validity of input data against specific criteria.

When the input data does not meet the criteria, the function uses `Brand.error` to generate a `BrandErrors` data type. This provides detailed information about why the validation failed.

```
import { Brand } from "effect"

// Define a branded type 'Int' to represent integer values
type Int = number & Brand.Brand<"Int">

// Define the constructor using 'refined' to enforce integer values
const Int = Brand.refined<Int>(
  // Validation to ensure the value is an integer
  (n) => Number.isInteger(n),
  // Provide an error if validation fails
  (n) => Brand.error(`Expected ${n} to be an integer`)
)
```

Usage example of the `Int` constructor:

```
import { Brand } from "effect"

type Int = number & Brand.Brand<"Int">

const Int = Brand.refined<Int>(
  (n) => Number.isInteger(n), // Check if the value is an integer
  (n) => Brand.error(`Expected ${n} to be an integer`) // Error message if the value is not an integer
)

// ---cut---
// Create a valid Int value
const x: Int = Int(3)
console.log(x) // Output: 3

// Attempt to create an Int with an invalid value
const y: Int = Int(3.14) // throws [ { message: 'Expected 3.14 to be an integer' } ]
```

Attempting to assign a non-`Int` value will result in a compile-time error:

```
// @errors: 2322
import { Brand } from "effect"

type Int = number & Brand.Brand<"Int">

const Int = Brand.refined<Int>(
  (n) => Number.isInteger(n),
  (n) => Brand.error(`Expected ${n} to be an integer`)
)

// ---cut---
// Correct usage
const good: Int = Int(3)

// Incorrect, will result in an error
const bad1: Int = 3
```

```
// Also incorrect, will result in an error
const bad2: Int = 3.14
```

Combining Branded Types

In some scenarios, you may need to combine multiple branded types together. The Brand module provides the `all` API to facilitate this:

```
import { Brand } from "effect"

type Int = number & Brand.Brand<"Int">

const Int = Brand.refined<Int>(
  (n) => Number.isInteger(n),
  (n) => Brand.error(`Expected ${n} to be an integer`)
)

type Positive = number & Brand.Brand<"Positive">

const Positive = Brand.refined<Positive>(
  (n) => n > 0,
  (n) => Brand.error(`Expected ${n} to be positive`)
)

// Combine the Int and Positive constructors into a new branded constructor PositiveInt
const PositiveInt = Brand.all(Int, Positive)

// Extract the branded type from the PositiveInt constructor
type PositiveInt = Brand.Brand.FromConstructor<typeof PositiveInt>

// Usage example

// Valid positive integer
const good: PositiveInt = PositiveInt(10)

// throws [ { message: 'Expected -5 to be positive' } ]
const bad1: PositiveInt = PositiveInt(-5)

// throws [ { message: 'Expected 3.14 to be an integer' } ]
const bad2: PositiveInt = PositiveInt(3.14)
```

Dual APIs

Location: 400-guides/720-style/100-dual

Explore "data-last" and "data-first" variants of dual APIs in the Effect ecosystem, illustrated with the example of `Effect.map`. Learn how to choose between them based on your coding style and readability preferences.

When you're working with APIs in the Effect ecosystem, you may come across two different ways to use the same API. These two ways are called the "data-last" and "data-first" variants.

<Info> From a technical perspective, these variants are implemented using two TypeScript overloads. </Info>

When an API supports both variants, we call them "dual" APIs.

Let's explore these two variants using a concrete example of a dual API: `Effect.map`.

The `Effect.map` function is defined with two TypeScript overloads. The terms "data-last" and "data-first" refer to the position of the `self` argument (also known as the "data") in the signatures of the two overloads:

```
export declare const map: {
  // data-last
  <A, B>(f: (a: A) => B): <E, R>(self: Effect<A, E, R>) => Effect<B, E, R>
  // data-first
  <A, E, R, B>(self: Effect<A, E, R>, f: (a: A) => B): Effect<B, E, R>
}
```

data-last

In the first overload, the `self` argument comes in the **last position**:

```
<A, B>(f: (a: A) => B): <E, R>(self: Effect<A, E, R>) => Effect<B, E, R>
```

This is the variant we have been using with `pipe`. You pass the `Effect` as the first argument to the `pipe` function, followed by a call to `Effect.andThen`:

```
const mappedEffect = pipe(effect, Effect.andThen(func))
```

This variant is useful when you need to chain multiple computations in a long pipeline. You can continue the pipeline by adding more computations after the initial transformation:

```
pipe(effect, Effect.andThen(func1), Effect.andThen(func2), ...)
```

data-first

In the second overload, the `self` argument comes in the **first position**:

```
<A, E, R, B>(self: Effect<A, E, R>, f: (a: A) => B): Effect<B, E, R>
```

This variant doesn't require the `pipe` function. Instead, you can directly pass the `Effect` as the first argument to the `Effect.andThen` function:

```
const mappedEffect = Effect.andThen(effect, func)
```

This variant is convenient when you only need to perform a single operation on the `Effect`.

Simplifying Excessive Nesting

Location: 400-guides/720-style/400-do

Learn how to simplify code with the `elapsed` function using different approaches. The guide demonstrates using plain pipe, the "do simulation," and the concise `Effect.gen` constructor to calculate and log the elapsed time for an effect's execution. Choose the method that fits your coding style and enhances code readability.

Suppose you want to create a custom function `elapsed` that prints the elapsed time taken by an effect to execute.

Using plain pipe

Initially, you may come up with code that uses the standard `pipe` method, but this approach can lead to excessive nesting and result in verbose and hard-to-read code:

```
import { Effect, Console } from "effect"

// Get the current timestamp
const now = Effect.sync(() => new Date().getTime())

// Prints the elapsed time occurred to `self` to execute
const elapsed = <R, E, A>(
  self: Effect.Effect<A, E, R>
): Effect.Effect<A, E, R> =>
  now.pipe(
    Effect.andThen((startMillis) =>
      self.pipe(
        Effect.andThen((result) =>
          now.pipe(
            Effect.andThen((endMillis) => {
              // Calculate the elapsed time in milliseconds
              const elapsed = endMillis - startMillis
              // Log the elapsed time
              return Console.log(`Elapsed: ${elapsed}`).pipe(
                Effect.map(() => result)
              )
            })
          )
        )
      )
    )
  )

// Simulates a successful computation with a delay of 200 milliseconds
const task = Effect.succeed("some task").pipe(Effect.delay("200 millis"))

const program = elapsed(task)

Effect.runPromise(program).then(console.log)
/*
Output:
Elapsed: 204
some task
*/
```

To address this issue and make the code more manageable, there is a solution: the "do simulation."

Using the "do simulation"

The "do simulation" in Effect allows you to write code in a more declarative style, similar to the "do notation" in other programming languages. It provides a way to define variables and perform operations on them using functions like `Effect.bind` and `Effect.let`.

Here's how the do simulation works:

1. Start the do simulation using the `Effect.Do` value:

   ```
   const program = Effect.Do.pipe(/* ... rest of the code */)
   ```

2. Within the do simulation scope, you can use the `Effect.bind` function to define variables and bind them to `Effect` values:

```
Effect.bind("variableName", (scope) => effectValue)
```

- `variableName` is the name you choose for the variable you want to define. It must be unique within the scope.
- `effectValue` is the `Effect` value that you want to bind to the variable. It can be the result of a function call or any other valid `Effect` value.

3. You can accumulate multiple `Effect.bind` statements to define multiple variables within the scope:

```
Effect.bind("variable1", () => effectValue1),
Effect.bind("variable2", ({ variable1 }) => effectValue2),
// ... additional bind statements
```

4. Inside the do simulation scope, you can also use the `Effect.let` function to define variables and bind them to simple values:

```
Effect.let("variableName", (scope) => simpleValue)
```

- `variableName` is the name you give to the variable. Like before, it must be unique within the scope.
- `simpleValue` is the value you want to assign to the variable. It can be a simple value like a `number`, `string`, or `boolean`.

5. Regular Effect functions like `Effect.andThen`, `Effect.flatMap`, `Effect.tap`, and `Effect.map` can still be used within the do simulation. These functions will receive the accumulated variables as arguments within the scope:

```
Effect.andThen(({ variable1, variable2 }) => {
  // Perform operations using variable1 and variable2
  // Return an `Effect` value as the result
})
```

With the do simulation, you can rewrite the `elapsed` combinator like this:

```
import { Effect, Console } from "effect"

// Get the current timestamp
const now = Effect.sync(() => new Date().getTime())

const elapsed = <R, E, A>(
  self: Effect.Effect<A, E, R>
): Effect.Effect<A, E, R> =>
  Effect.Do.pipe(
    Effect.bind("startMillis", () => now),
    Effect.bind("result", () => self),
    Effect.bind("endMillis", () => now),
    Effect.let(
      "elapsed",
      ({ startMillis, endMillis }) => endMillis - startMillis // Calculate the elapsed time in milliseconds
    ),
    Effect.tap(({ elapsed }) => Console.log(`Elapsed: ${elapsed}`)), // Log the elapsed time
    Effect.map(({ result }) => result)
  )

// Simulates a successful computation with a delay of 200 milliseconds
const task = Effect.succeed("some task").pipe(Effect.delay("200 millis"))

const program = elapsed(task)

Effect.runPromise(program).then(console.log)
/*
Output:
Elapsed: 204
some task
*/
```

In this solution, we use the do simulation to simplify the code. The `elapsed` function now starts with `Effect.Do` to enter the simulation scope. Inside the scope, we use `Effect.bind` to define variables and bind them to the corresponding effects.

Using `Effect.gen`

The most concise and convenient solution is to use the `Effect.gen` constructor, which allows you to work with generators when dealing with effects. This approach leverages the native scope provided by the generator syntax, avoiding excessive nesting and leading to more concise code.

```
import { Effect } from "effect"

// Get the current timestamp
const now = Effect.sync(() => new Date().getTime())

// Prints the elapsed time occurred to `self` to execute
const elapsed = <R, E, A>(
  self: Effect.Effect<A, E, R>
): Effect.Effect<A, E, R> =>
  Effect.gen(function* () {
    const startMillis = yield* now
    const result = yield* self
    const endMillis = yield* now
    // Calculate the elapsed time in milliseconds
    const elapsed = endMillis - startMillis
    // Log the elapsed time
```

```
    console.log(`Elapsed: ${elapsed}`)
    return result
  })

// Simulates a successful computation with a delay of 200 milliseconds
const task = Effect.succeed("some task").pipe(Effect.delay("200 millis"))

const program = elapsed(task)

Effect.runPromise(program).then(console.log)
/*
Output:
Elapsed: 204
some task
*/
```

In this solution, we switch to using generators to simplify the code. The `elapsed` function now uses a generator function (`Effect.gen`) to define the flow of execution. Within the generator, we use `yield*` to invoke effects and bind their results to variables. This eliminates the nesting and provides a more readable and sequential code structure.

The generator style in Effect uses a more linear and sequential flow of execution, resembling traditional imperative programming languages. This makes the code easier to read and understand, especially for developers who are more familiar with imperative programming paradigms.

On the other hand, the pipe style can lead to excessive nesting, especially when dealing with complex effectful computations. This can make the code harder to follow and debug.

For more information on how to use generators in Effect, you can refer to the Using Generators in Effect guide.

Code Style

Location: 400-guides/720-style/index

Code Style

Guidelines

Location: 400-guides/720-style/500-guidelines

Guidelines

Using `runMain`

In Effect, `runMain` serves as the primary entry point for running an Effect application on Node:

```
import { Effect, Console, Schedule, pipe } from "effect"
import { NodeRuntime } from "@effect/platform-node"

const program = pipe(
  Effect.addFinalizer(() => Console.log("Application is about to exit!")),
  Effect.andThen(Console.log("Application started!")),
  Effect.andThen(
    Effect.repeat(Console.log("still alive..."), {
      schedule: Schedule.spaced("1 second")
    })
  ),
  Effect.scoped
)

// Effect.runFork(program) // no graceful teardown with CTRL+C

NodeRuntime.runMain(program) // graceful teardown with CTRL+C
```

The `runMain` function is responsible for finding all fibers and interrupting them. Internally, it adds an observer for the fiber by listening to `sigint` and interrupts all fibers.

It's important to note that teardown should be on the main effect. If you kill the fiber that runs the application/server, the teardown of everything will occur. This is precisely what `runMain` from the `platform-node` package does.

Avoid Tacit usage

> Avoid tacit function calls like `map(f)` and using `flow`

In Effect, it's recommended not to use functions point-free, meaning avoiding tacit usage.

While you're free to use tacit functions if you prefer, it's important to know that it can cause issues. It's safer to use `(x) => fn(x)` instead.

Using functions tacitly, especially with optional parameters, can be unsafe. If a function has overloads, using it tacitly might erase all generics, leading to bugs. For example, check out this X thread: link to thread.

TypeScript inference can also be compromised when using tacit functions, which can lead to unexpected errors. So, it's not just a matter of style; it's a protective measure to avoid mistakes.

Additionally, stack traces may not be as clear when tacit usage is involved. It's risky without much benefit, as TypeScript may not properly check arguments, especially with optional ones, leading to potential issues.

It's worth trying without tacit usage, especially when dealing with generic functions with overloads, as using them tacitly can result in losing the generics.

Pattern Matching

Location: 400-guides/720-style/300-pattern-matching

Explore the power of pattern matching in code, simplifying complex conditions into concise expressions. Learn about exhaustiveness checking and discover how to implement pattern matching in JavaScript using the `effect/Match` module. Dive into defining matchers, patterns, predicates, and transformations for enhanced code branching and readability.

Pattern matching is a method that allows developers to handle intricate conditions within a single, concise expression. It simplifies code, making it more concise and easier to understand. Additionally, it includes a process called exhaustiveness checking, which helps to ensure that no possible case has been overlooked.

Originating from functional programming languages, pattern matching stands as a powerful technique for code branching. It often offers a more potent and less verbose solution compared to imperative alternatives such as if/else or switch statements, particularly when dealing with complex conditions.

Although not yet a native feature in JavaScript, there's an ongoing tc39 proposal in its early stages to introduce pattern matching to JavaScript. However, this proposal is at stage 1 and might take several years to be implemented. Nonetheless, developers can implement pattern matching in their codebase. The `effect/Match` module provides a reliable, type-safe pattern matching implementation that is available for immediate use.

Defining a Matcher

type

Creating a `Matcher` involves using the `type` constructor function with a specified type. This sets the foundation for pattern matching against that particular type. Once the `Matcher` is established, developers can employ various combinators like `when`, `not`, and `tag` to define patterns that the `Matcher` will check against.

Here's a practical example:

```
import { Match } from "effect"

const match = Match.type<{ a: number } | { b: string }>().pipe(
  Match.when({ a: Match.number }, (_) => _.a),
  Match.when({ b: Match.string }, (_) => _.b),
  Match.exhaustive
)

console.log(match({ a: 0 })) // Output: 0
console.log(match({ b: "hello" })) // Output: "hello"
```

Let's dissect what's happening:

- `Match.type<{ a: number } | { b: string }>()`: This creates a `Matcher` for objects that are either of type `{ a: number }` or `{ b: string }`.
- `Match.when({ a: Match.number }, (_) => _.a)`: This sets up a condition to match an object with a property `a` containing a number. If matched, it returns the value of property `a`.
- `Match.when({ b: Match.string }, (_) => _.b)`: This condition matches an object with a property `b` containing a string. If found, it returns the value of property `b`.
- `Match.exhaustive`: This function ensures that all possible cases are considered and matched, making sure that no other unaccounted cases exist. It helps to prevent overlooking any potential scenario.

Finally, the `match` function is applied to test two different objects, `{ a: 0 }` and `{ b: "hello" }`. As per the defined conditions within the `Matcher`, it correctly matches the objects and provides the expected output based on the defined conditions.

value

In addition to defining a `Matcher` based on a specific type, developers can also create a `Matcher` directly from a value utilizing the `value` constructor function. This method allows matching patterns against the provided value.

Let's take a look at an example to better understand this process:

```
import { Match } from "effect"

const result = Match.value({ name: "John", age: 30 }).pipe(
  Match.when(
    { name: "John" },
    (user) => `${user.name} is ${user.age} years old`
```

```
  ),
  Match.orElse(() => "Oh, not John")
)

console.log(result) // Output: "John is 30 years old"
```

Here's a breakdown of what's happening:

- `Match.value({ name: "John", age: 30 })`: This initializes a `Matcher` using the provided value `{ name: "John", age: 30 }`.
- `Match.when({ name: "John" }, (user) => ...)`: It establishes a condition to match the object having the property `name` set to "John". If the condition is met, it constructs a string indicating the name and age of the user.
- `Match.orElse(() => "Oh, not John")`: In the absence of a match with the name "John," this provides a default output.

Patterns

Predicates

Predicates allow the testing of values against specific conditions. It helps in creating rules or conditions for the data being evaluated.

```
import { Match } from "effect"

const match = Match.type<{ age: number }>().pipe(
  Match.when({ age: (age) => age >= 5 }, (user) => `Age: ${user.age}`),
  Match.orElse((user) => `${user.age} is too young`)
)

console.log(match({ age: 5 })) // Output: "Age: 5"
console.log(match({ age: 4 })) // Output: "4 is too young"
```

not

`not` allows for excluding a specific value while matching other conditions.

```
import { Match } from "effect"

const match = Match.type<string | number>().pipe(
  Match.not("hi", (_) => "a"),
  Match.orElse(() => "b")
)

console.log(match("hello")) // Output: "a"
console.log(match("hi")) // Output: "b"
```

tag

The `tag` function enables pattern matching against the tag within a Discriminated Union.

```
import { Match, Either } from "effect"

const match = Match.type<Either.Either<number, string>>().pipe(
  Match.tag("Right", (_) => _.right),
  Match.tag("Left", (_) => _.left),
  Match.exhaustive
)

console.log(match(Either.right(123))) // Output: 123
console.log(match(Either.left("Oh no!"))) // Output: "Oh no!"
```

<Warning> Note that it only works with the convention within the Effect ecosystem of naming the tag field with `"_tag"`. </Warning>

Transforming a Matcher

exhaustive

The `exhaustive` transformation serves as an endpoint within the matching process, ensuring all potential matches have been considered. It results in returning the match (for `Match.value`) or the evaluation function (for `Match.type`).

```
import { Match, Either } from "effect"

const result = Match.value(Either.right(0)).pipe(
  Match.when({ _tag: "Right" }, (_) => _.right),
  // @ts-expect-error
  Match.exhaustive // TypeError! Type 'Left<never, number>' is not assignable to type 'never'
)
```

orElse

The `orElse` transformation signifies the conclusion of the matching process, offering a fallback value when no specific patterns match. It returns the match (for `Match.value`) or the evaluation function (for `Match.type`).

```
import { Match } from "effect"

const match = Match.type<string | number>().pipe(
  Match.when("hi", (_) => "hello"),
  Match.orElse(() => "I literally do not understand")
)

console.log(match("hi")) // Output: "hello"
console.log(match("hello")) // Output: "I literally do not understand"
```

option

The `option` transformation returns the result encapsulated within an Option. When the match succeeds, it represents the result as `Some`, and when there's no match, it signifies the absence of a value with `None`.

```
import { Match, Either } from "effect"

const result = Match.value(Either.right(0)).pipe(
  Match.when({ _tag: "Right" }, (_) => _.right),
  Match.option
)

console.log(result) // Output: { _id: 'Option', _tag: 'Some', value: 0 }
```

either

The `either` transformation might match a value, returning an Either following the format `Either<MatchResult, NoMatchResult>`.

```
import { Match } from "effect"

const match = Match.type<string>().pipe(
  Match.when("hi", (_) => _.length),
  Match.either
)

console.log(match("hi")) // Output: { _id: 'Either', _tag: 'Right', right: 2 }
console.log(match("shigidigi")) // Output: { _id: 'Either', _tag: 'Left', left: 'shigidigi' }
```

Running Effects

Location: 400-guides/100-essentials/400-running-effects

Explore various "run" functions in the Effect module to execute effects. Learn about `runSync` for synchronous execution, `runSyncExit` for obtaining results as `Exit`, `runPromise` for executing with a Promise result, and `runPromiseExit` for Promise results with `Exit`. Understand their use cases and considerations. Check out a cheatsheet summarizing available functions for executing effects in different contexts.

To execute an `Effect`, we can utilize a variety of "run" functions provided by the `Effect` module.

runSync

The `Effect.runSync` function is used to execute an Effect synchronously, which means it runs immediately and returns the result.

```
import { Effect } from "effect"

const program = Effect.sync(() => {
  console.log("Hello, World!")
  return 1
})

const result = Effect.runSync(program)
// Output: Hello, World!

console.log(result)
// Output: 1
```

If you check the console, you will see the message `"Hello, World!"` printed.

<Warning> `Effect.runSync` will throw an error if your Effect fails or performs any asynchronous tasks. In the latter case, the execution will not proceed beyond that asynchronous task. </Warning>

```
import { Effect } from "effect"

Effect.runSync(Effect.fail("my error")) // throws

Effect.runSync(Effect.promise(() => Promise.resolve(1))) // throws
```

runSyncExit

The `Effect.runSyncExit` function is used to execute an Effect synchronously, which means it runs immediately and returns the result as an Exit (a data type used to describe the result of executing an `Effect` workflow).

```
import { Effect } from "effect"

const result1 = Effect.runSyncExit(Effect.succeed(1))
console.log(result1)
/*
Output:
{
  _id: "Exit",
  _tag: "Success",
  value: 1
}
*/

const result2 = Effect.runSyncExit(Effect.fail("my error"))
console.log(result2)
/*
Output:
{
  _id: "Exit",
  _tag: "Failure",
  cause: {
    _id: "Cause",
    _tag: "Fail",
    failure: "my error"
  }
}
*/
```

<Warning> `Effect.runSyncExit` will throw an error if your Effect performs any asynchronous tasks and the execution will not proceed beyond that asynchronous task. </Warning>

```
import { Effect } from "effect"

Effect.runSyncExit(Effect.promise(() => Promise.resolve(1))) // throws
```

runPromise

The `Effect.runPromise` function is used to execute an Effect and obtain the result as a `Promise`.

```
import { Effect } from "effect"

Effect.runPromise(Effect.succeed(1)).then(console.log) // Output: 1
```

<Warning> `Effect.runPromise` will reject with an error if your Effect fails </Warning>

```
import { Effect } from "effect"

Effect.runPromise(Effect.fail("my error")) // rejects
```

runPromiseExit

The `Effect.runPromiseExit` function is used to execute an Effect and obtain the result as a `Promise` that resolves to an Exit (a data type used to describe the result of executing an `Effect` workflow).

```
import { Effect } from "effect"

Effect.runPromiseExit(Effect.succeed(1)).then(console.log)
/*
Output:
{
  _id: "Exit",
  _tag: "Success",
  value: 1
}
*/

Effect.runPromiseExit(Effect.fail("my error")).then(console.log)
/*
Output:
{
  _id: "Exit",
  _tag: "Failure",
  cause: {
    _id: "Cause",
    _tag: "Fail",
    failure: "my error"
  }
}
*/
```

runFork

The `Effect.runFork` function serves as a foundational building block for running effects. In fact, all other run functions are built upon it. Unless you have a specific need for a Promise or a synchronous operation, `Effect.runFork` is the recommended choice. It returns a fiber that you can

observe or interrupt as needed.

```
import { Effect, Console, Schedule, Fiber } from "effect"

const program = Effect.repeat(
  Console.log("running..."),
  Schedule.spaced("200 millis")
)

const fiber = Effect.runFork(program)

setTimeout(() => {
  Effect.runFork(Fiber.interrupt(fiber))
}, 500)
```

In this example, the `program` continuously logs "running..." with each repetition spaced 200 milliseconds apart. You can learn more about repetitions and scheduling in our Introduction to Scheduling guide.

To stop the execution of the program, we use `Fiber.interrupt` on the fiber returned by `Effect.runFork`. This allows you to control the execution flow and terminate it when necessary.

For a deeper understanding of how fibers work and how to handle interruptions, check out our guides on Fibers and Interruptions.

Cheatsheet

The table provides a summary of the available `run*` functions, along with their input and output types, allowing you to choose the appropriate function based on your needs.

Name	Given	To
runSync	Effect<A, E>	A
runSyncExit	Effect<A, E>	Exit<A, E>
runPromise	Effect<A, E>	Promise<A>
runPromiseExit	Effect<A, E>	Promise<Exit<A, E>>
runFork	Effect<A, E>	RuntimeFiber<A, E>

You can find the complete list of `run*` functions here.

Creating Effects

Location: 400-guides/100-essentials/300-creating-effects

Learn various methods to create effects in the Effect ecosystem. Understand the drawbacks of throwing errors in traditional programming and explore constructors like `Effect.succeed` and `Effect.fail` for explicit success and failure handling. Dive into modeling synchronous effects with `Effect.sync` and `Effect.try`, and asynchronous effects with `Effect.promise` and `Effect.tryPromise`. Explore `Effect.async` for callback-based APIs and `Effect.suspend` for deferred effect evaluation. Check out a cheatsheet summarizing available constructors.

Effect provides different ways to create effects, which are units of computation that encapsulate side effects. In this guide, we will cover some of the common methods that you can use to create effects.

Why Not Throw Errors?

In traditional programming, when an error occurs, it is often handled by throwing an exception:

```
const divide = (a: number, b: number): number => {
  if (b === 0) {
    throw new Error("Cannot divide by zero")
  }
  return a / b
}
```

However, throwing errors can be problematic. The type signatures of functions do not indicate that they can throw exceptions, making it difficult to reason about potential errors.

To address this issue, Effect introduces dedicated constructors for creating effects that represent both success and failure: `Effect.succeed` and `Effect.fail`. These constructors allow you to explicitly handle success and failure cases while **leveraging the type system to track errors**.

succeed

The `Effect.succeed` constructor in the Effect library is used to explicitly create an effect that is guaranteed to succeed. Here's how you can use it:

```
import { Effect } from "effect"

const success = Effect.succeed(42)
```

In this example, `success` is an instance of `Effect<number, never, never>`. This means it's an effect that:

- Always succeeds, yielding a value of type `number`.

- Does not generate any errors (`never` indicates that no errors are expected).
- Requires no additional data or dependencies from the environment (`never` indicates no requirements).

fail

When a computation might fail, it's essential to manage the failure explicitly. The `Effect.fail` constructor allows you to encapsulate an error within your program flow explicitly. This method is useful for representing known error states in a predictable and type-safe way. Here's a practical example to illustrate:

```
import { Effect } from "effect"

// Creating an effect that represents a failure scenario
const failure = Effect.fail(
  new Error("Operation failed due to network error")
)
```

The type of `failure` is `Effect<never, Error, never>`, which means

- It never produces a successful value (`never`).
- It fails with an error, specifically an `Error`.
- It does not depend on any external context to execute (`never`).

You're not limited to using only `Error` objects with `Effect.fail`. You can also use strings, numbers, or more complex objects depending on what fits best with your error handling strategy:

```
import { Effect } from "effect"

const failure = Effect.fail("Something went wrong")
```

With `Effect.succeed` and `Effect.fail`, you can explicitly handle success and failure cases and the type system will ensure that errors are tracked and accounted for.

Example: Rewriting a Division Function

Let's see an example of rewriting the `divide` function using Effect to make the error handling explicit:

```
import { Effect } from "effect"

const divide = (a: number, b: number): Effect.Effect<number, Error> =>
  b === 0
    ? Effect.fail(new Error("Cannot divide by zero"))
    : Effect.succeed(a / b)
```

In this example, the `divide` function explicitly indicates that it can produce an effect that either fails with an `Error` or succeeds with a `number` value. The type signature makes it clear how errors are handled and ensures that callers are aware of the possible outcomes.

Example: Simulating a User Retrieval Operation

Let's imagine another scenario where we use `Effect.succeed` and `Effect.fail` to model a simple user retrieval operation where the user data is hardcoded, which could be useful in testing scenarios or when mocking data:

```
import { Effect } from "effect"

// Define a User type
interface User {
  readonly id: number
  readonly name: string
}

// A mocked function to simulate fetching a user from a database
const getUser = (userId: number): Effect.Effect<User, Error> => {
  // Normally, you would access a database or an API here, but we'll mock it
  const userDatabase: Record<number, User> = {
    1: { id: 1, name: "John Doe" },
    2: { id: 2, name: "Jane Smith" }
  }

  // Check if the user exists in our "database" and return appropriately
  const user = userDatabase[userId]
  if (user) {
    return Effect.succeed(user)
  } else {
    return Effect.fail(new Error("User not found"))
  }
}

// When executed, this will successfully return the user with id 1
const exampleUserEffect = getUser(1)
```

In this example `exampleUserEffect` can result in either a `User` object or an `Error`, depending on whether the user exists in the simulated database

To dive deeper into handling and managing errors effectively in your applications using Effect, you might want to explore the guide on Error Management. This guide provides detailed insights and strategies for robust error handling in TypeScript applications using Effect.

Modeling Synchronous Effects

In JavaScript, you can delay the execution of synchronous computations using "thunks".

<Info> A "thunk" is a function that takes no arguments and may return some value. </Info>

Thunks are useful for delaying the computation of a value until it is needed.

To model synchronous side effects, Effect provides the `Effect.sync` and `Effect.try` constructors, which accept a thunk.

sync

When working with side effects that are synchronous — meaning they don't involve asynchronous operations like fetching data from the internet — you can use the `Effect.sync` function. This function is ideal when you are certain these operations **won't produce any errors**.

Example: Logging a Message

```
import { Effect } from "effect"

const log = (message: string) =>
  Effect.sync(() => {
    console.log(message) // side effect
  })

const program = log("Hello, World!")
```

In the above example, `Effect.sync` is used to defer the side-effect of writing to the console.

The `program` has the type `Effect<void, never, never>`, indicating that:

- It doesn't produce a return value (`void`).
- It's not expected to fail (`never` indicates no expected errors).
- It doesn't require any external dependencies or context (`never`).

Important Notes:

- **Execution**: The side effect (logging to the console) encapsulated within `program` won't occur until the effect is explicitly run. This allows you to define side effects at one point in your code and control when they are activated, improving manageability and predictability of side effects in larger applications.
- **Error Handling**: It's crucial that the function you pass to `Effect.sync` does not throw any errors. If you anticipate potential errors, consider using try instead, which handles errors gracefully.

<Error>The thunk passed to `Effect.sync` should never throw errors.</Error>

Handling Unexpected Errors. Despite your best efforts to avoid errors in the function passed to `Effect.sync`, if an error does occur, it results in a "defect". This defect is not a standard error but indicates a flaw in the logic that was expected to be error-free. You can think of it similar to an unexpected crash in the program, which can be further managed or logged using tools like Effect.catchAllDefect. This feature ensures that even unexpected failures in your application are not lost and can be handled appropriately.

try

In situations where you need to perform synchronous operations that might fail, such as parsing JSON, you can use the `Effect.try` constructor from the Effect library. This constructor is designed to handle operations that could throw exceptions by capturing those exceptions and transforming them into manageable errors within the Effect framework.

Example: Safe JSON Parsing

Suppose you have a function that attempts to parse a JSON string. This operation can fail and throw an error if the input string is not properly formatted as JSON:

```
import { Effect } from "effect"

const parse = (input: string) =>
  Effect.try(
    () => JSON.parse(input) // This might throw an error if input is not valid JSON
  )

const program = parse("")
```

In this example:

- `parse` is a function that creates an effect encapsulating the JSON parsing operation.
- If `JSON.parse(input)` throws an error due to invalid input, `Effect.try` catches this error and the effect represented by `program` will fail with an `UnknownException`. This ensures that errors are not silently ignored but are instead handled within the structured flow of effects.

Customizing Error Handling. You might want to transform the caught exception into a more specific error or perform additional operations when catching an error. `Effect.try` supports an overload that allows you to specify how caught exceptions should be transformed:

Example: Custom Error Handling

```
import { Effect } from "effect"

const parse = (input: string) =>
  Effect.try({
    try: () => JSON.parse(input), // JSON.parse may throw for bad input
    catch: (unknown) => new Error(`something went wrong ${unknown}`) // remap the error
  })

const program = parse("")
```

You can think of this as a similar pattern to the traditional try-catch block in JavaScript:

```
try {
  return JSON.parse(input)
} catch (unknown) {
  throw new Error(`something went wrong ${unknown}`)
}
```

Modeling Asynchronous Effects

In traditional programming, we often use `Promise`s to handle asynchronous computations. However, dealing with errors in promises can be problematic. By default, `Promise<Value>` only provides the type `Value` for the resolved value, which means errors are not reflected in the type system. This limits the expressiveness and makes it challenging to handle and track errors effectively.

To overcome these limitations, Effect introduces dedicated constructors for creating effects that represent both success and failure in an asynchronous context: `Effect.promise` and `Effect.tryPromise`. These constructors allow you to explicitly handle success and failure cases while **leveraging the type system to track errors**.

promise

This constructor is similar to a regular `Promise`, where you're confident that the asynchronous operation will **always succeed**. It allows you to create an `Effect` that represents successful completion without considering potential errors. However, it's essential to ensure that the underlying Promise never rejects.

Example: Delayed Message

```
import { Effect } from "effect"

const delay = (message: string) =>
  Effect.promise<string>(
    () =>
      new Promise((resolve) => {
        setTimeout(() => {
          resolve(message)
        }, 2000)
      })
  )

const program = delay("Async operation completed successfully!")
```

The `program` value has the type `Effect<string, never, never>` and can be interpreted as an effect that:

- succeeds with a value of type `string`
- does not produce any expected error (`never`)
- does not require any context (`never`)

<Error> The `Promise` within the thunk passed to `Effect.promise` should never reject. </Error>

Handling Unexpected Errors. If, despite precautions, the thunk passed to `Effect.promise` does reject, an `Effect` containing a "defect" is created, similar to what happens when using the Effect.die function.

tryPromise

Unlike `Effect.promise`, this constructor is suitable when the underlying `Promise` **might reject**. It provides a way to catch errors and handle them appropriately. By default if an error occurs, it will be caught and propagated to the error channel as as an `UnknownException`.

Example: Fetching a TODO Item

```
import { Effect } from "effect"

const getTodo = (id: number) =>
  Effect.tryPromise(() =>
    fetch(`https://jsonplaceholder.typicode.com/todos/${id}`)
  )

const program = getTodo(1)
```

The `program` value has the type `Effect<Response, UnknownException, never>` and can be interpreted as an effect that:

- succeeds with a value of type `Response`
- might produce an error (`UnknownException`)
- does not require any context (`never`)

Customizing Error Handling. If you want more control over what gets propagated to the error channel, you can use an overload of `Effect.tryPromise` that takes a remapping function:

```
import { Effect } from "effect"

const getTodo = (id: number) =>
  Effect.tryPromise({
    try: () => fetch(`https://jsonplaceholder.typicode.com/todos/${id}`),
    // remap the error
    catch: (unknown) => new Error(`something went wrong ${unknown}`)
  })

const program = getTodo(1)
```

From a callback

Sometimes you have to work with APIs that don't support `async/await` or `Promise` and instead use the callback style. To handle callback-based APIs, Effect provides the `Effect.async` constructor.

Example: Reading a File

For example, let's wrap the `readFile` async API from the Node.js `fs` module with Effect (ensure you have `@types/node` installed):

```
// @types: node
import { Effect } from "effect"
import * as NodeFS from "node:fs"

const readFile = (filename: string) =>
  Effect.async<Buffer, Error>((resume) => {
    NodeFS.readFile(filename, (error, data) => {
      if (error) {
        resume(Effect.fail(error))
      } else {
        resume(Effect.succeed(data))
      }
    })
  })

const program = readFile("todos.txt")
```

In the above example, we manually annotate the types when calling `Effect.async` because TypeScript cannot infer the type parameters for a callback based on the return value inside the callback body. Annotating the types ensures that the values provided to `resume` match the expected types.

<Idea> You can seamlessly mix synchronous and asynchronous code within the Effect framework. Everything becomes an Effect, enabling you to handle different types of side effects in a unified way. </Idea>

Suspended Effects

`Effect.suspend` is used to delay the creation of an effect. It allows you to defer the evaluation of an effect until it is actually needed. The `Effect.suspend` function takes a thunk that represents the effect, and it wraps it in a suspended effect.

```
const suspendedEffect = Effect.suspend(() => effect)
```

Let's explore some common scenarios where `Effect.suspend` proves useful:

1. **Lazy Evaluation**. When you want to defer the evaluation of an effect until it is required. This can be useful for optimizing the execution of effects, especially when they are not always needed or when their computation is expensive.

   Also, when effects with side effects or scoped captures are created, use `Effect.suspend` to re-execute on each invocation.

   ```
   import { Effect } from "effect"

   let i = 0

   const bad = Effect.succeed(i++)

   const good = Effect.suspend(() => Effect.succeed(i++))

   console.log(Effect.runSync(bad)) // Output: 0
   console.log(Effect.runSync(bad)) // Output: 0

   console.log(Effect.runSync(good)) // Output: 1
   console.log(Effect.runSync(good)) // Output: 2
   ```

   In this example, `Effect.succeed(i++)` creates a new numeric value and consistently returns the same number. On the other hand, `Effect.suspend(() => Effect.succeed(i++))` generates a new number with each invocation.

   This example utilizes `Effect.runSync` to execute effects and display their results (refer to Running Effects for more details).

2. **Handling Circular Dependencies**. `Effect.suspend` is helpful in managing circular dependencies between effects, where one effect depends on another, and vice versa. For example it's fairly common for `Effect.suspend` to be used in recursive functions to escape an eager call. For

instance:

```
import { Effect } from "effect"

const blowsUp = (n: number): Effect.Effect<number> =>
  n < 2
    ? Effect.succeed(1)
    : Effect.zipWith(blowsUp(n - 1), blowsUp(n - 2), (a, b) => a + b)

// console.log(Effect.runSync(blowsUp(32))) // crash: JavaScript heap out of memory

const allGood = (n: number): Effect.Effect<number> =>
  n < 2
    ? Effect.succeed(1)
    : Effect.zipWith(
        Effect.suspend(() => allGood(n - 1)),
        Effect.suspend(() => allGood(n - 2)),
        (a, b) => a + b
      )

console.log(Effect.runSync(allGood(32))) // Output: 3524578
```

3. **Unifying Return Type**. In situations where TypeScript struggles to unify the returned effect type, `Effect.suspend` can be employed to resolve this issue. For example:

```
import { Effect } from "effect"

const ugly = (a: number, b: number) =>
  b === 0
    ? Effect.fail(new Error("Cannot divide by zero"))
    : Effect.succeed(a / b)

const nice = (a: number, b: number) =>
  Effect.suspend(() =>
    b === 0
      ? Effect.fail(new Error("Cannot divide by zero"))
      : Effect.succeed(a / b)
  )
```

Cheatsheet

The table provides a summary of the available constructors, along with their input and output types, allowing you to choose the appropriate function based on your needs.

Function	Given	To
`succeed`	`A`	`Effect<A>`
`fail`	`E`	`Effect<never, E>`
`sync`	`() => A`	`Effect<A>`
`try`	`() => A`	`Effect<A, UnknownException>`
`try` (overload)	`() => A`, `unknown => E`	`Effect<A, E>`
`promise`	`() => Promise<A>`	`Effect<A>`
`tryPromise`	`() => Promise<A>`	`Effect<A, UnknownException>`
`tryPromise` (overload)	`() => Promise<A>`, `unknown => E`	`Effect<A, E>`
`async`	`(Effect<A, E> => void) => void`	`Effect<A, E>`
`suspend`	`() => Effect<A, E, R>`	`Effect<A, E, R>`

You can find the complete list of constructors here.

Now that we know how to create effects, it's time to learn how to run them. Check out the next guide on Running Effects to find out more.

Building Pipelines

Location: 400-guides/100-essentials/600-pipeline

Explore the power of Effect pipelines for composing and sequencing operations on values. Learn about key functions like `pipe`, `Effect.map`, `Effect.flatMap`, `Effect.andThen`, `Effect.tap`, and `Effect.all` for building modular and concise transformations. Understand the advantages of using functions over methods in the Effect ecosystem for tree shakeability and extensibility.

Effect pipelines allow for the composition and sequencing of operations on values, enabling the transformation and manipulation of data in a concise and modular manner.

Why Pipelines are Good for Structuring Your Application

Pipelines are an excellent way to structure your application and handle data transformations in a concise and modular manner. They offer several benefits:

1. **Readability**: Pipelines allow you to compose functions in a readable and sequential manner. You can clearly see the flow of data and the operations applied to it, making it easier to understand and maintain the code.

2. **Code Organization**: With pipelines, you can break down complex operations into smaller, manageable functions. Each function performs a specific task, making your code more modular and easier to reason about.

3. **Reusability**: Pipelines promote the reuse of functions. By breaking down operations into smaller functions, you can reuse them in different pipelines or contexts, improving code reuse and reducing duplication.

4. **Type Safety**: By leveraging the type system, pipelines help catch errors at compile-time. Functions in a pipeline have well-defined input and output types, ensuring that the data flows correctly through the pipeline and minimizing runtime errors.

Now, let's delve into how to define pipelines and explore some of the key components:

pipe

The `pipe` function is a utility that allows us to compose functions in a readable and sequential manner. It takes the output of one function and passes it as the input to the next function in the pipeline. This enables us to build complex transformations by chaining multiple functions together.

The basic syntax of `pipe` is as follows:

```
import { pipe } from "effect"

const result = pipe(input, func1, func2, ..., funcN)
```

In this syntax, `input` is the initial value, and `func1`, `func2`, ..., `funcN` are the functions to be applied in sequence. The result of each function becomes the input for the next function, and the final result is returned.

Here's an illustration of how `pipe` works:

It's important to note that functions passed to `pipe` must have a **single argument** because they are only called with a single argument.

Let's see an example to better understand how `pipe` works:

```
import { pipe } from "effect"

// Define simple arithmetic operations
const increment = (x: number) => x + 1
const double = (x: number) => x * 2
const subtractTen = (x: number) => x - 10

// Sequentially apply these operations using `pipe`
const result = pipe(5, increment, double, subtractTen)

console.log(result) // Output: 2
```

In the above example, we start with an input value of `5`. The `increment` function adds `1` to the initial value, resulting in `6`. Then, the `double` function doubles the value, giving us `12`. Finally, the `subtractTen` function subtracts `10` from `12`, resulting in the final output of `2`.

The result is equivalent to `subtractTen(double(increment(5)))`, but using `pipe` makes the code more readable because the operations are sequenced from left to right, rather than nesting them inside out.

Functions vs Methods

In the Effect ecosystem, libraries often expose functions rather than methods. This design choice is important for two key reasons: tree shakeability and extensibility.

Tree Shakeability

Tree shakeability refers to the ability of a build system to eliminate unused code during the bundling process. Functions are tree shakeable, while methods are not.

When functions are used in the Effect ecosystem, only the functions that are actually imported and used in your application will be included in the final bundled code. Unused functions are automatically removed, resulting in a smaller bundle size and improved performance.

On the other hand, methods are attached to objects or prototypes, and they cannot be easily tree shaken. Even if you only use a subset of methods, all methods associated with an object or prototype will be included in the bundle, leading to unnecessary code bloat.

Extensibility

Another important advantage of using functions in the Effect ecosystem is the ease of extensibility. With methods, extending the functionality of an existing API often requires modifying the prototype of the object, which can be complex and error-prone.

In contrast, with functions, extending the functionality is much simpler. You can define your own "extension methods" as plain old functions without the need to modify the prototypes of objects. This promotes cleaner and more modular code, and it also allows for better compatibility with other libraries and modules.

<Idea> The use of functions in the Effect ecosystem libraries is important for achieving **tree shakeability** and ensuring **extensibility**. Functions enable efficient bundling by eliminating unused code, and they provide a flexible and modular approach to extending the libraries' functionality.

</Idea>

Now let's explore some examples of APIs that can be used with the `pipe` function to build pipelines.

map

The `Effect.map` function is used to transform the value inside an `Effect`. It takes a function and applies it to the value contained within the `Effect`, creating a **new** `Effect` with the transformed value.

Usage of Effect.map

The syntax for `Effect.map` is as follows:

```
import { pipe, Effect } from "effect"

const mappedEffect = pipe(myEffect, Effect.map(transformation))
// or
const mappedEffect = Effect.map(myEffect, transformation)
// or
const mappedEffect = myEffect.pipe(Effect.map(transformation))
```

In the code above, `transformation` is the function applied to the value, and `myEffect` is the `Effect` being transformed.

<Info> It's important to note that `Effect`s are immutable, meaning that when you use `Effect.map` on an `Effect`, it doesn't modify the original data type. Instead, it returns a new copy of the `Effect` with the transformed value. </Info>

Example

Consider a program that adds a small service charge to a transaction:

```
import { pipe, Effect } from "effect"

// Function to add a small service charge to a transaction amount
const addServiceCharge = (amount: number) => amount + 1

// Simulated asynchronous task to fetch a transaction amount from a database
const fetchTransactionAmount = Effect.promise(() => Promise.resolve(100))

// Apply service charge to the transaction amount
const finalAmount = pipe(fetchTransactionAmount, Effect.map(addServiceCharge))

Effect.runPromise(finalAmount).then(console.log) // Output: 101
```

as

To map an `Effect` to a constant value, replacing the original value, use `Effect.as`:

```
import { pipe, Effect } from "effect"

const program = pipe(Effect.succeed(5), Effect.as("new value"))

Effect.runPromise(program).then(console.log) // Output: "new value"
```

flatMap

The `Effect.flatMap` function is used when you need to chain transformations that produce `Effect` instances. This is useful for asynchronous operations or computations that depend on the results of previous effects.

Usage of Effect.flatMap

The `Effect.flatMap` function enables you to sequence computations that result in new `Effect` values, "flattening" any nested `Effect` structures that arise.

The syntax for `Effect.flatMap` is as follows:

```
import { pipe, Effect } from "effect"

const flatMappedEffect = pipe(myEffect, Effect.flatMap(transformation))
// or
const flatMappedEffect = Effect.flatMap(myEffect, transformation)
// or
const flatMappedEffect = myEffect.pipe(Effect.flatMap(transformation))
```

In the code above, `transformation` is the function that takes a value and returns an `Effect`, and `myEffect` is the initial `Effect` being transformed.

<Info> It's important to note that `Effect`s are immutable, meaning that when you use `Effect.flatMap` on an `Effect`, it doesn't modify the original data type. Instead, it returns a new copy of the `Effect` with the transformed value. </Info>

Example

```
import { pipe, Effect } from "effect"

// Function to apply a discount safely to a transaction amount
const applyDiscount = (
  total: number,
  discountRate: number
): Effect.Effect<number, Error> =>
  discountRate === 0
    ? Effect.fail(new Error("Discount rate cannot be zero"))
    : Effect.succeed(total - (total * discountRate) / 100)

// Simulated asynchronous task to fetch a transaction amount from a database
const fetchTransactionAmount = Effect.promise(() => Promise.resolve(100))

const finalAmount = pipe(
  fetchTransactionAmount,
  Effect.flatMap((amount) => applyDiscount(amount, 5))
)

Effect.runPromise(finalAmount).then(console.log) // Output: 95
```

Ensuring All Effects Are Considered

It's vital to ensure that all effects within `Effect.flatMap` contribute to the final computation. Neglecting any effect can lead to unexpected behaviors or incorrect outcomes:

```
Effect.flatMap((amount) => {
  Effect.sync(() => console.log(`Apply a discount to: ${amount}`)) // This effect is ignored
  return applyDiscount(amount, 5)
})
```

The `Effect.sync` above is ignored and does not influence the result of `applyDiscount(amount, 5)`. To include effects properly and avoid errors, explicitly chain them using functions like `Effect.map`, `Effect.flatMap`, `Effect.andThen`, or `Effect.tap`.

Further Information on `flatMap`

Although many developers may recognize `flatMap` from its usage with arrays, in the `Effect` framework, it's utilized to manage and resolve nested `Effect` structures. If your goal is to flatten nested arrays within an Effect (`Effect<Array<Array<A>>>`), this can be done using:

```
import { pipe, Effect, Array } from "effect"

const flattened = pipe(
  Effect.succeed([
    [1, 2],
    [3, 4]
  ]),
  Effect.map((nested) => Array.flatten(nested))
)
```

or using the standard `Array.prototype.flat()` method.

andThen

Both the `Effect.map` and `Effect.flatMap` functions serve to transform an `Effect` into another `Effect` in two different scenarios. In the first scenario, `Effect.map` is used when the transformation function does not return an `Effect`, while in the second scenario, `Effect.flatMap` is used when the transformation function still returns an `Effect`. However, since both scenarios involve transformations, the Effect module also exposes a convenient all-in-one solution to use: `Effect.andThen`.

The `Effect.andThen` function executes a sequence of two actions, typically two `Effect`s, where the second action can depend on the result of the first action.

```
import { pipe, Effect } from "effect"

const transformedEffect = pipe(myEffect, Effect.andThen(anotherEffect))
// or
const transformedEffect = Effect.andThen(myEffect, anotherEffect)
// or
const transformedEffect = myEffect.pipe(Effect.andThen(anotherEffect))
```

The `anotherEffect` action can take various forms:

- a value
- a function returning a value (i.e. same functionality of `Effect.map`)
- a `Promise`
- a function returning a `Promise`
- an `Effect`
- a function returning an `Effect`(i.e. same functionality of `Effect.flatMap`)

Example

Let's see an example where we can compare the use of `Effect.andThen` instead of `Effect.map` and `Effect.flatMap`:

```
import { pipe, Effect } from "effect"
```

```
// Function to apply a discount safely to a transaction amount
const applyDiscount = (
  total: number,
  discountRate: number
): Effect.Effect<number, Error> =>
  discountRate === 0
    ? Effect.fail(new Error("Discount rate cannot be zero"))
    : Effect.succeed(total - (total * discountRate) / 100)

// Simulated asynchronous task to fetch a transaction amount from a database
const fetchTransactionAmount = Effect.promise(() => Promise.resolve(100))

// Using Effect.map, Effect.flatMap
const result1 = pipe(
  fetchTransactionAmount,
  Effect.map((amount) => amount * 2),
  Effect.flatMap((amount) => applyDiscount(amount, 5))
)

Effect.runPromise(result1).then(console.log) // Output: 190

// Using Effect.andThen
const result2 = pipe(
  fetchTransactionAmount,
  Effect.andThen((amount) => amount * 2),
  Effect.andThen((amount) => applyDiscount(amount, 5))
)

Effect.runPromise(result2).then(console.log) // Output: 190
```

tap

The `Effect.tap` API has a similar signature to `Effect.flatMap`, but the result of the transformation function is **ignored**. This means that the value returned by the previous computation will still be available for the next computation.

Example

```
import { pipe, Effect } from "effect"

// Function to apply a discount safely to a transaction amount
const applyDiscount = (
  total: number,
  discountRate: number
): Effect.Effect<number, Error> =>
  discountRate === 0
    ? Effect.fail(new Error("Discount rate cannot be zero"))
    : Effect.succeed(total - (total * discountRate) / 100)

// Simulated asynchronous task to fetch a transaction amount from a database
const fetchTransactionAmount = Effect.promise(() => Promise.resolve(100))

const finalAmount = pipe(
  fetchTransactionAmount,
  Effect.tap((amount) =>
    Effect.sync(() => console.log(`Apply a discount to: ${amount}`))
  ),
  // `amount` is still available!
  Effect.flatMap((amount) => applyDiscount(amount, 5))
)

Effect.runPromise(finalAmount).then(console.log)
/*
Output:
Apply a discount to: 100
95
*/
```

Using `Effect.tap` allows us to execute side effects during the computation without altering the result. This can be useful for logging, performing additional actions, or observing the intermediate values without interfering with the main computation flow.

all

The `Effect.all` function is a powerful utility provided by Effect that allows you to combine multiple effects into a single effect that produces a tuple of results.

Usage of Effect.all

The syntax for `Effect.all` is as follows:

```
import { Effect } from "effect"

const combinedEffect = Effect.all([effect1, effect2, ...])
```

The `Effect.all` function will execute all these effects in **sequence** (to explore options for managing concurrency and controlling how these effects are executed, you can refer to the Concurrency Options documentation).

It will return a new effect that produces a tuple containing the results of each individual effect. Keep in mind that the order of the results corresponds to the order of the original effects passed to `Effect.all`.

Example

```
import { Effect } from "effect"

// Simulated function to read configuration from a file
const webConfig = Effect.promise(() =>
  Promise.resolve({ dbConnection: "localhost", port: 8080 })
)

// Simulated function to test database connectivity
const checkDatabaseConnectivity = Effect.promise(() =>
  Promise.resolve("Connected to Database")
)

// Combine both effects to perform startup checks
const startupChecks = Effect.all([webConfig, checkDatabaseConnectivity])

Effect.runPromise(startupChecks).then(([config, dbStatus]) => {
  console.log(
    `Configuration: ${JSON.stringify(config)}, DB Status: ${dbStatus}`
  )
})
/*
Output:
Configuration: {"dbConnection":"localhost","port":8080}, DB Status: Connected to Database
*/
```

<Info> The `Effect.all` function not only combines tuples but also works with iterables, structs, and records. To explore the full potential of `all` head over to the Introduction to Effect's Control Flow Operators documentation. </Info>

Build your first pipeline

Now, let's combine `pipe`, `Effect.all` and `Effect.andThen` to build a pipeline that performs a series of transformations:

```
import { Effect, pipe } from "effect"

// Function to add a small service charge to a transaction amount
const addServiceCharge = (amount: number) => amount + 1

// Function to apply a discount safely to a transaction amount
const applyDiscount = (
  total: number,
  discountRate: number
): Effect.Effect<number, Error> =>
  discountRate === 0
    ? Effect.fail(new Error("Discount rate cannot be zero"))
    : Effect.succeed(total - (total * discountRate) / 100)

// Simulated asynchronous task to fetch a transaction amount from a database
const fetchTransactionAmount = Effect.promise(() => Promise.resolve(100))

// Simulated asynchronous task to fetch a discount rate from a configuration file
const fetchDiscountRate = Effect.promise(() => Promise.resolve(5))

// Assembling the program using a pipeline of effects
const program = pipe(
  Effect.all([fetchTransactionAmount, fetchDiscountRate]),
  Effect.flatMap(([transactionAmount, discountRate]) =>
    applyDiscount(transactionAmount, discountRate)
  ),
  Effect.map(addServiceCharge),
  Effect.map((finalAmount) => `Final amount to charge: ${finalAmount}`)
)

// Execute the program and log the result
Effect.runPromise(program).then(console.log) // Output: "Final amount to charge: 96"
```

The pipe method

Effect provides a `pipe` method that works similarly to the `pipe` method found in rxjs. This method allows you to chain multiple operations together, making your code more concise and readable.

Here's how the `pipe` **method** works:

```
const result = effect.pipe(func1, func2, ..., funcN)
```

This is equivalent to using the `pipe` **function** like this:

```
const result = pipe(effect, func1, func2, ..., funcN)
```

The `pipe` method is available on all effects and many other data types, eliminating the need to import the `pipe` function from the `Function` module and saving you some keystrokes.

Let's rewrite the previous example using the `pipe` method:

```
import { Effect } from "effect"

const addServiceCharge = (amount: number) => amount + 1

const applyDiscount = (
  total: number,
  discountRate: number
): Effect.Effect<number, Error> =>
  discountRate === 0
    ? Effect.fail(new Error("Discount rate cannot be zero"))
    : Effect.succeed(total - (total * discountRate) / 100)

const fetchTransactionAmount = Effect.promise(() => Promise.resolve(100))

const fetchDiscountRate = Effect.promise(() => Promise.resolve(5))

// ---cut---
const program = Effect.all([fetchTransactionAmount, fetchDiscountRate]).pipe(
  Effect.flatMap(([transactionAmount, discountRate]) =>
    applyDiscount(transactionAmount, discountRate)
  ),
  Effect.map(addServiceCharge),
  Effect.map((finalAmount) => `Final amount to charge: ${finalAmount}`)
)
```

Cheatsheet

Let's summarize the transformation functions we have seen so far:

Function	Input	Output
`map`	`Effect<A, E, R>, A => B`	`Effect<B, E, R>`
`flatMap`	`Effect<A, E, R>, A => Effect<B, E, R>`	`Effect<B, E, R>`
`andThen`	`Effect<A, E, R>, *`	`Effect<B, E, R>`
`tap`	`Effect<A, E, R>, A => Effect<B, E, R>`	`Effect<A, E, R>`
`all`	`[Effect<A, E, R>, Effect<B, E, R>, ...]`	`Effect<[A, B, ...], E, R>`

These functions are powerful tools for transforming and chaining `Effect` computations. They allow you to apply functions to values inside `Effect` and build complex pipelines of computations.

The Effect Type

Location: 400-guides/100-essentials/200-the-effect-type

Explore the `Effect` type in the Effect ecosystem, representing an immutable, lazy description of a workflow or job. Understand its type parameters and conceptualize it as an effectful program. Learn how to interpret `Effect` values with the Effect Runtime System for effectful interactions with the external world.

The `Effect<Success, Error, Requirements>` type represents an **immutable** value that **lazily** describes a workflow or job.

This type encapsulates the logic of a program, defining whether it succeeds, providing a value of type `Success`, or fails, resulting in an error of type `Error`. Additionally, the program requires a collection of contextual data `Context<Requirements>` to execute.

Conceptually, you can think of `Effect<Success, Error, Requirements>` as an effectful version of the following function type:

```
type Effect<Success, Error, Requirements> = (
  context: Context<Requirements>
) => Error | Success
```

However, effects are not actually functions. They can model synchronous, asynchronous, concurrent, and resourceful computations.

Type Parameters

The `Effect` type has three type parameters with the following meanings:

- **Success**. Represents the type of value that an effect can succeed with when executed. If this type parameter is `void`, it means the effect produces no useful information, while if it is `never`, it means the effect runs forever (or until failure).
- **Error**. Represents the expected errors that can occur when executing an effect. If this type parameter is `never`, it means the effect cannot fail, because there are no values of type `never`.
- **Requirements**. Represents the contextual data required by the effect to be executed. This data is stored in a collection named `Context`. If this type parameter is `never`, it means the effect has no requirements and the `Context` collection is empty.

<Info> In the Effect ecosystem, you may often encounter the type parameters of `Effect` abbreviated as `A`, `E`, and `R` respectively. This is just shorthand for the success value of type **A**, **E**rror, and **R**equirements. </Info>

`Effect` values are immutable, and all Effect functions produce new `Effect` values.

`Effect` values do not actually do anything, they are just values that model or describe effectful interactions.

An `Effect` can be interpreted by the Effect Runtime System into effectful interactions with the external world. Ideally, this occurs at a single entry point in our application where the effectful interactions are initiated, such as the starting point of your program's execution.

Effect Essentials

Location: 400-guides/100-essentials/index

Effect Essentials

Using Generators in Effect

Location: 400-guides/100-essentials/500-using-generators

Explore the syntax of using generators in Effect to write effectful code. Learn about the `Effect.gen` function. Compare `Effect.gen` with `async`/`await` for writing asynchronous code. Understand how generators enhance control flow, handle errors, and utilize short-circuiting in effectful programs. Discover passing references to `this` in generator functions.

In the previous sections, we learned how to create effects and execute them. Now, it's time to write our first simple program.

Effect offers a convenient syntax, similar to `async`/`await`, to write effectful code using generators.

<Idea> The use of generators is an **optional feature** in Effect. If you find generators unfamiliar or prefer a different coding style, you can explore the documentation about Building Pipelines in Effect. </Idea>

Understanding Effect.gen

The `Effect.gen` utility simplifies the task of writing effectful code by utilizing JavaScript's generator functions. This method helps your code appear and behave more like traditional synchronous code, which enhances both readability and error management.

Let's explore a practical program that performs a series of data transformations commonly found in application logic:

```
import { Effect } from "effect"

// Function to add a small service charge to a transaction amount
const addServiceCharge = (amount: number) => amount + 1

// Function to apply a discount safely to a transaction amount
const applyDiscount = (
  total: number,
  discountRate: number
): Effect.Effect<number, Error> =>
  discountRate === 0
    ? Effect.fail(new Error("Discount rate cannot be zero"))
    : Effect.succeed(total - (total * discountRate) / 100)

// Simulated asynchronous task to fetch a transaction amount from a database
const fetchTransactionAmount = Effect.promise(() => Promise.resolve(100))

// Simulated asynchronous task to fetch a discount rate from a configuration file
const fetchDiscountRate = Effect.promise(() => Promise.resolve(5))

// Assembling the program using a generator function
const program = Effect.gen(function* () {
  // Retrieve the transaction amount
  const transactionAmount = yield* fetchTransactionAmount

  // Retrieve the discount rate
  const discountRate = yield* fetchDiscountRate

  // Calculate discounted amount
  const discountedAmount = yield* applyDiscount(
    transactionAmount,
    discountRate
  )

  // Apply service charge
  const finalAmount = addServiceCharge(discountedAmount)

  // Return the total amount after applying the charge
  return `Final amount to charge: ${finalAmount}`
})

// Execute the program and log the result
Effect.runPromise(program).then(console.log) // Output: "Final amount to charge: 96"
```

Key steps to follow when using `Effect.gen`:

- Wrap your logic in `Effect.gen`

- Use `yield*` to handle effects
- Return the final result

<Warning> The generator API is only available when using the `downlevelIteration` flag or with a `target` of `"es2015"` or higher in your `tsconfig.json` file </Warning>

Comparing Effect.gen with async/await

If you are familiar with `async`/`await`, you may notice that the flow of writing code is similar.

Let's compare the two approaches:

It's important to note that although the code appears similar, the two programs are not identical. The purpose of comparing them side by side is just to highlight the resemblance in how they are written.

Embracing Control Flow

One significant advantage of using `Effect.gen` in conjunction with generators is its capability to employ standard control flow constructs within the generator function. These constructs include `if`/`else`, `for`, `while`, and other branching and looping mechanisms, enhancing your ability to express complex control flow logic in your code.

```
import { Effect } from "effect"

const calculateTax = (
  amount: number,
  taxRate: number
): Effect.Effect<number, Error> =>
  taxRate > 0
    ? Effect.succeed((amount * taxRate) / 100)
    : Effect.fail(new Error("Invalid tax rate"))

const program = Effect.gen(function* () {
  let i = 1

  while (true) {
    if (i === 10) {
      break // Break the loop when counter reaches 10
    } else {
      if (i % 2 === 0) {
        // Calculate tax for even numbers
        console.log(yield* calculateTax(100, i))
      }
      i++
      continue
    }
  }
})

Effect.runPromise(program)
/*
Output:
2
4
6
8
*/
```

Raising Errors

The `Effect.gen` API allows you to incorporate error handling directly into your program flow by yielding failed effects. This mechanism, achieved through `Effect.fail`, is demonstrated in the example below:

```
import { Effect } from "effect"

const program = Effect.gen(function* () {
  console.log("Task1...")
  console.log("Task2...")
  // Introduce an error into the flow
  yield* Effect.fail("Something went wrong!")
})

Effect.runPromiseExit(program).then(console.log)
/*
Output:
Task1...
Task2...
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: 'Something went wrong!' }
}
*/
```

The Role of Short-Circuiting

When working with the `Effect.gen` API, it's important to understand how it manages errors. This API is designed to **short-circuit the execution** upon encountering the **first error**.

What does this mean for you as a developer? Well, let's say you have a chain of operations or a collection of effects to be executed in sequence. If any error occurs during the execution of one of these effects, the remaining computations will be skipped, and the error will be propagated to the final result.

In simpler terms, the short-circuiting behavior ensures that if something goes wrong at any step of your program it will immediately stop and return the error to let you know that something went wrong.

```
import { Effect } from "effect"

const program = Effect.gen(function* () {
  console.log("Task1...")
  console.log("Task2...")
  yield* Effect.fail("Something went wrong!")
  console.log("This won't be executed")
})

Effect.runPromise(program).then(console.log, console.error)
/*
Output:
Task1...
Task2...
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: 'Something went wrong!' }
}
*/
```

<Info> If you want to dive deeper into effective error handling with Effect, you can explore the "Error Management" section. </Info>

Passing this

In some cases, you might need to pass a reference to the current object (`this`) into the body of your generator function. You can achieve this by utilizing an overload that accepts the reference as the first argument:

```
import { Effect } from "effect"

class MyService {
  readonly local = 1
  compute = Effect.gen(this, function* () {
    return yield* Effect.succeed(this.local + 1)
  })
}

console.log(Effect.runSync(new MyService().compute)) // Output: 2
```

Importing Effect

Location: 400-guides/100-essentials/100-importing-effect

Start using Effect by installing the `effect` package and exploring module imports. Learn how to import the `Effect` module, understand namespace imports, and grasp the advantages of using functions over methods in the Effect ecosystem. Focus on essential functions to build a strong foundation for your journey with Effect.

Welcome to Effect! If you're just getting started, you might feel overwhelmed by the variety of modules and functions that Effect offers. However, rest assured that you don't need to worry about all of them right away. In this guide, we will provide you with a simple orientation on how to import modules and functions and assure you that, in most cases, installing the `effect` package is all you need to get started. Let's dive in!

Installing Effect

To begin your journey with Effect, you first need to install the `effect` package. Open your terminal and run the following command:

By installing this single package, you gain access to the core functionality of Effect.

For more detailed installation instructions on other platforms like Deno or Bun, you can refer to the Quickstart tutorial. This will provide you with step-by-step guidance to set up Effect on different environments.

Importing Modules and Functions

Once you have installed the `effect` package, you can start using its modules and functions in your projects. Importing modules and functions is straightforward and follows the standard JavaScript/TypeScript import syntax.

To import a module or a function from the `effect` package, simply use the `import` statement at the top of your file. Here's how you can import the `Effect` module:

```
import { Effect } from "effect"
```

Now, you have access to the `Effect` module, which is the heart of the Effect library. It provides various functions to create, compose, and manipulate effectful computations.

Namespace imports

In addition to importing the `Effect` module with a named import, as shown previously:

```
import { Effect } from "effect"
```

You can also import it using a namespace import like this:

```
import * as Effect from "effect/Effect"
```

Both forms of import allow you to access the functionalities provided by the `Effect` module.

However an important consideration is **tree shaking**, which refers to a process that eliminates unused code during the bundling of your application. Named imports may generate tree shaking issues when a bundler doesn't support deep scope analysis.

Here are some bundlers that support deep scope analysis and thus don't have issues with named imports:

- Rollup
- Webpack 5+

Functions vs Methods

In the Effect ecosystem, libraries often expose functions rather than methods. This design choice is important for two key reasons: tree shakeability and extendibility.

Tree Shakeability

Tree shakeability refers to the ability of a build system to eliminate unused code during the bundling process. Functions are tree shakeable, while methods are not.

When functions are used in the Effect ecosystem, only the functions that are actually imported and used in your application will be included in the final bundled code. Unused functions are automatically removed, resulting in a smaller bundle size and improved performance.

On the other hand, methods are attached to objects or prototypes, and they cannot be easily tree shaken. Even if you only use a subset of methods, all methods associated with an object or prototype will be included in the bundle, leading to unnecessary code bloat.

Extendibility

Another important advantage of using functions in the Effect ecosystem is the ease of extendibility. With methods, extending the functionality of an existing API often requires modifying the prototype of the object, which can be complex and error-prone.

In contrast, with functions, extending the functionality is much simpler. You can define your own "extension methods" as plain old functions without the need to modify the prototypes of objects. This promotes cleaner and more modular code, and it also allows for better compatibility with other libraries and modules.

<Idea> The use of functions in the Effect ecosystem libraries is important for achieving **tree shakeability** and ensuring **extendibility**. Functions enable efficient bundling by eliminating unused code, and they provide a flexible and modular approach to extending the libraries' functionality. </Idea>

Commonly Used Functions

As you start your adventure with Effect, you don't need to dive into every function in the `effect` package right away. Instead, focus on some commonly used functions that will provide a solid foundation for your journey into the world of Effect.

In the upcoming guides, we will explore some of these essential functions, specifically those for creating and running `Effect`s and building pipelines.

But before we dive into those, let's start from the very heart of Effect: understanding the `Effect` type. This will lay the groundwork for your understanding of how Effect brings composability, type safety, and error handling into your applications.

So, let's take the first step and explore the fundamental concepts of the The Effect Type.

Batching & Caching

Location: 400-guides/620-batching-caching

Effect is a powerful TypeScript library designed to help developers easily create complex, synchronous, and asynchronous programs.

Classic Approach to API Integration

In typical application development, when interacting with external APIs, databases, or other data sources, we often define functions that perform requests and handle their results or failures accordingly.

Simple Model Setup

Here's a basic model that outlines the structure of our data and possible errors:

```
export interface User {
  readonly _tag: "User"
  readonly id: number
  readonly name: string
  readonly email: string
}

export class GetUserError {
  readonly _tag = "GetUserError"
}

export interface Todo {
  readonly _tag: "Todo"
  readonly id: number
  readonly message: string
  readonly ownerId: number
}

export class GetTodosError {
  readonly _tag = "GetTodosError"
}

export class SendEmailError {
  readonly _tag = "SendEmailError"
}

// @include: Model
```

<Idea> In a real world scenario we may want to use a more precise types instead of directly using primitives for identifiers (see Branded Types). Additionally, you may want to include more detailed information in the errors. </Idea>

Defining API Functions

Let's define functions that interact with an external API, handling common operations such as fetching todos, retrieving user details, and sending emails.

```
import { Effect } from "effect"
import * as Model from "./Model"

// Fetches a list of todos from an external API
export const getTodos = Effect.tryPromise({
  try: () =>
    fetch("https://api.example.demo/todos").then(
      (res) => res.json() as Promise<Array<Model.Todo>>
    ),
  catch: () => new Model.GetTodosError()
})

// Retrieves a user by their ID from an external API
export const getUserById = (id: number) =>
  Effect.tryPromise({
    try: () =>
      fetch(`https://api.example.demo/getUserById?id=${id}`).then(
        (res) => res.json() as Promise<Model.User>
      ),
    catch: () => new Model.GetUserError()
  })

// Sends an email via an external API
export const sendEmail = (address: string, text: string) =>
  Effect.tryPromise({
    try: () =>
      fetch("https://api.example.demo/sendEmail", {
        method: "POST",
        headers: {
          "Content-Type": "application/json"
        },
        body: JSON.stringify({ address, text })
      }).then((res) => res.json() as Promise<void>),
    catch: () => new Model.SendEmailError()
  })

// Sends an email to a user by fetching their details first
export const sendEmailToUser = (id: number, message: string) =>
  getUserById(id).pipe(
```

```
    Effect.andThen((user) => sendEmail(user.email, message))
  )

// Notifies the owner of a todo by sending them an email
export const notifyOwner = (todo: Model.Todo) =>
  getUserById(todo.ownerId).pipe(
    Effect.andThen((user) =>
      sendEmailToUser(user.id, `hey ${user.name} you got a todo!`)
    )
  )

// @filename: Model.ts
// @include: Model

// @filename: API.ts
// ---cut---
// @include: API
```

<Idea> In a real-world scenario, you might not want to trust your APIs to always return the expected data - for this, you can use `@effect/schema` or similar alternatives such as `zod`. </Idea>

While this approach is straightforward and readable, it may not be the most efficient. Repeated API calls, especially when many todos share the same owner, can significantly increase network overhead and slow down your application.

Using the API Functions

While these functions are clear and easy to understand, their use may not be the most efficient. For example, notifying todo owners involves repeated API calls which can be optimized.

```
// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import * as API from "./API"

// Orchestrates operations on todos, notifying their owners
const program = Effect.gen(function* () {
  const todos = yield* API.getTodos
  yield* Effect.forEach(todos, (todo) => API.notifyOwner(todo), {
    concurrency: "unbounded"
  })
})
```

This implementation performs an API call for each todo to fetch the owner's details and send an email. If multiple todos have the same owner, this results in redundant API calls.

Improving Efficiency with Batch Calls

To optimize, consider implementing batch API calls if your backend supports them. This reduces the number of HTTP requests by grouping multiple operations into a single request, thereby enhancing performance and reducing load.

Next Steps:

Refactor your API interactions to use batch processing where possible. This not only reduces server load but also streamlines the handling of data, keeping your code both efficient and clean.

Batching

Batching API calls can drastically improve the performance of your application by reducing the number of HTTP requests.

Let's assume that `getUserById` and `sendEmail` can be batched. This means that we can send multiple requests in a single HTTP call, reducing the number of API requests and improving performance.

Step-by-Step Guide to Batching

1. **Structuring Requests:** We'll start by transforming our requests into structured data models. This involves detailing input parameters, expected outputs, and possible errors. Structuring requests this way not only helps in efficiently managing data but also in comparing different requests to understand if they refer to the same input parameters.

2. **Defining Resolvers:** Resolvers are designed to handle multiple requests simultaneously. By leveraging the ability to compare requests (ensuring they refer to the same input parameters), resolvers can execute several requests in one go, maximizing the utility of batching.

3. **Creating Queries:** Finally, we'll define queries that utilize these batch-resolvers to perform operations. This step ties together the structured requests and their corresponding resolvers into functional components of the application.

Important Considerations

It's crucial for the requests to be modeled in a way that allows them to be comparable. This means implementing comparability (using methods like Equals.equals) to identify and batch identical requests effectively.

Declaring Requests

Let's start by defining a structured model for the types of requests our data sources can handle. We'll design a model using the concept of a `Request` that a data source might support.

A `Request<Value, Error>` is a construct representing a request for a value of type `Value`, which might fail with an error of type `Error`.

```
import { Request } from "effect"
import * as Model from "./Model"

// Define a request to get multiple Todo items which might fail with a GetTodosError
export interface GetTodos
  extends Request.Request<Array<Model.Todo>, Model.GetTodosError> {
  readonly _tag: "GetTodos"
}

// Create a tagged constructor for GetTodos requests
export const GetTodos = Request.tagged<GetTodos>("GetTodos")

// Define a request to fetch a User by ID which might fail with a GetUserError
export interface GetUserById
  extends Request.Request<Model.User, Model.GetUserError> {
  readonly _tag: "GetUserById"
  readonly id: number
}

// Create a tagged constructor for GetUserById requests
export const GetUserById = Request.tagged<GetUserById>("GetUserById")

// Define a request to send an email which might fail with a SendEmailError
export interface SendEmail
  extends Request.Request<void, Model.SendEmailError> {
  readonly _tag: "SendEmail"
  readonly address: string
  readonly text: string
}

// Create a tagged constructor for SendEmail requests
export const SendEmail = Request.tagged<SendEmail>("SendEmail")

// Combine all requests into a union type for easier management
export type ApiRequest = GetTodos | GetUserById | SendEmail
```

```
// @filename: Model.ts
// @include: Model

// @filename: Requests.ts
// ---cut---
// @include: Requests
```

Each request is defined with a specific data structure that extends from a generic `Request` type, ensuring that each request carries its unique data requirements along with a specific error type.

By using tagged constructors like `Request.tagged`, we can easily instantiate request objects that are recognizable and manageable throughout the application.

Declaring Resolvers

After defining our requests, the next step is configuring how Effect resolves these requests using `RequestResolver`. A `RequestResolver<A, R>` requires an environment `R` and is capable of executing requests of type `A`.

In this section, we'll create individual resolvers for each type of request. The granularity of your resolvers can vary, but typically, they are divided based on the batching capabilities of the corresponding API calls.

```
import { Effect, RequestResolver, Request } from "effect"
import * as API from "./API"
import * as Model from "./Model"
import * as Requests from "./Requests"

// Assuming GetTodos cannot be batched, we create a standard resolver.
export const GetTodosResolver = RequestResolver.fromEffect(
  (request: Requests.GetTodos) => API.getTodos
)

// Assuming GetUserById can be batched, we create a batched resolver.
export const GetUserByIdResolver = RequestResolver.makeBatched(
  (requests: ReadonlyArray<Requests.GetUserById>) =>
    Effect.tryPromise({
      try: () =>
        fetch("https://api.example.demo/getUserByIdBatch", {
          method: "POST",
          headers: {
            "Content-Type": "application/json"
```

```
          },
          body: JSON.stringify({
            users: requests.map(({ id }) => ({ id }))
          })
        }).then((res) => res.json()) as Promise<Array<Model.User>>,
      catch: () => new Model.GetUserError()
    }).pipe(
      Effect.andThen((users) =>
        Effect.forEach(requests, (request, index) =>
          Request.completeEffect(request, Effect.succeed(users[index]))
        )
      ),
      Effect.catchAll((error) =>
        Effect.forEach(requests, (request) =>
          Request.completeEffect(request, Effect.fail(error))
        )
      )
    )
)

// Assuming SendEmail can be batched, we create a batched resolver.
export const SendEmailResolver = RequestResolver.makeBatched(
  (requests: ReadonlyArray<Requests.SendEmail>) =>
    Effect.tryPromise({
      try: () =>
        fetch("https://api.example.demo/sendEmailBatch", {
          method: "POST",
          headers: {
            "Content-Type": "application/json"
          },
          body: JSON.stringify({
            emails: requests.map(({ address, text }) => ({ address, text }))
          })
        }).then((res) => res.json() as Promise<void>),
      catch: () => new Model.SendEmailError()
    }).pipe(
      Effect.andThen(
        Effect.forEach(requests, (request) =>
          Request.completeEffect(request, Effect.void)
        )
      ),
      Effect.catchAll((error) =>
        Effect.forEach(requests, (request) =>
          Request.completeEffect(request, Effect.fail(error))
        )
      )
    )
)

// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: Resolvers.ts
// ---cut---
// @include: Resolvers
```

<Info> Resolvers can also access the context like any other `Effect`, and there are many different ways to create resolvers. For further details, consider exploring the reference documentation for the RequestResolver module. </Info>

In this configuration:

- **GetTodosResolver** handles the fetching of multiple Todo items. It's set up as a standard resolver since we assume it cannot be batched.
- **GetUserByIdResolver** and **SendEmailResolver** are configured as batched resolvers. This setup is based on the assumption that these requests can be processed in batches, enhancing performance and reducing the number of API calls.

Defining Queries

Now that we've set up our resolvers, we're ready to tie all the pieces together to define our queries. This step will enable us to perform data operations effectively within our application.

```
import { Effect } from "effect"
import * as Model from "./Model"
import * as Requests from "./Requests"
import * as Resolvers from "./Resolvers"

// Defines a query to fetch all Todo items
export const getTodos: Effect.Effect<
  Array<Model.Todo>,
  Model.GetTodosError
> = Effect.request(Requests.GetTodos({}), Resolvers.GetTodosResolver)

// Defines a query to fetch a user by their ID
```

```ts
export const getUserById = (id: number) =>
  Effect.request(
    Requests.GetUserById({ id }),
    Resolvers.GetUserByIdResolver
  )

// Defines a query to send an email to a specific address
export const sendEmail = (address: string, text: string) =>
  Effect.request(
    Requests.SendEmail({ address, text }),
    Resolvers.SendEmailResolver
  )

// Composes getUserById and sendEmail to send an email to a specific user
export const sendEmailToUser = (id: number, message: string) =>
  getUserById(id).pipe(
    Effect.andThen((user) => sendEmail(user.email, message))
  )

// Uses getUserById to fetch the owner of a Todo and then sends them an email notification
export const notifyOwner = (todo: Model.Todo) =>
  getUserById(todo.ownerId).pipe(
    Effect.andThen((user) =>
      sendEmailToUser(user.id, `hey ${user.name} you got a todo!`)
    )
  )

// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: Resolvers.ts
// @include: Resolvers

// @filename: Queries.ts
// ---cut---
// @include: Queries
```

By using the `Effect.request` function, we integrate the resolvers with the request model effectively. This approach ensures that each query is optimally resolved using the appropriate resolver.

Although the code structure looks similar to earlier examples, employing resolvers significantly enhances efficiency by optimizing how requests are handled and reducing unnecessary API calls.

```ts
// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: Resolvers.ts
// @include: Resolvers

// @filename: Queries.ts
// @include: Queries

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import * as Queries from "./Queries"

const program = Effect.gen(function* () {
  const todos = yield* Queries.getTodos
  yield* Effect.forEach(todos, (todo) => Queries.notifyOwner(todo), {
    batching: true
  })
})
```

In the final setup, this program will execute only **3** queries to the APIs, regardless of the number of todos. This contrasts sharply with the traditional approach, which would potentially execute **1 + 2n** queries, where **n** is the number of todos. This represents a significant improvement in efficiency, especially for applications with a high volume of data interactions.

Disabling Batching

Batching can be locally disabled using the `Effect.withRequestBatching` utility in the following way:

```ts
// @filename: Model.ts
// @include: Model

// @filename: API.ts
```

```
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: Resolvers.ts
// @include: Resolvers

// @filename: Queries.ts
// @include: Queries

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import * as Queries from "./Queries"

const program = Effect.gen(function* () {
  const todos = yield* Queries.getTodos
  yield* Effect.forEach(todos, (todo) => Queries.notifyOwner(todo), {
    concurrency: "unbounded"
  })
}).pipe(Effect.withRequestBatching(false))
```

Resolvers with Context

In complex applications, resolvers often need access to shared services or configurations to handle requests effectively. However, maintaining the ability to batch requests while providing the necessary context can be challenging. Here, we'll explore how to manage context in resolvers to ensure that batching capabilities are not compromised.

When creating request resolvers, it's crucial to manage the context carefully. Providing too much context or providing varying services to resolvers can make them incompatible for batching. To prevent such issues, the context for the resolver used in `Effect.request` is explicitly set to `never`. This forces developers to clearly define how the context is accessed and used within resolvers.

Consider the following example where we set up an HTTP service that the resolvers can use to execute API calls:

```
import { Effect, Context, RequestResolver } from "effect"
import * as Model from "./Model"
import * as Requests from "./Requests"

export class HttpService extends Context.Tag("HttpService")<
  HttpService,
  { fetch: typeof fetch }
>() {}

export const GetTodosResolver =
  // we create a normal resolver like we did before
  RequestResolver.fromEffect((request: Requests.GetTodos) =>
    Effect.andThen(HttpService, (http) =>
      Effect.tryPromise({
        try: () =>
          http
            .fetch("https://api.example.demo/todos")
            .then((res) => res.json() as Promise<Array<Model.Todo>>),
        catch: () => new Model.GetTodosError()
      })
    )
  ).pipe(
    // we list the tags that the resolver can access
    RequestResolver.contextFromServices(HttpService)
  )

// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: ResolversWithContext.ts
// ---cut---
// @include: ResolversWithContext
```

We can see now that the type of `GetTodosResolver` is no longer a `RequestResolver` but instead it is:

```
Effect<RequestResolver<GetTodos, never>, never, HttpService>
```

which is an `Effect` that access the `HttpService` and returns a composed resolver that has the minimal context ready to use.

Once we have such `Effect` we can directly use it in our query definition:

```
// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API
```

```
// @filename: Requests.ts
// @include: Requests

// @filename: ResolversWithContext.ts
// @include: ResolversWithContext

// @filename: QueriesWithContext.ts
// ---cut---
import { Effect } from "effect"
import * as Model from "./Model"
import * as Requests from "./Requests"
import * as ResolversWithContext from "./ResolversWithContext"

export const getTodos = Effect.request(
  Requests.GetTodos({}),
  ResolversWithContext.GetTodosResolver
)
```

We can see that the Effect correctly requires `HttpService` to be provided.

Alternatively you can create `RequestResolver`s as part of layers direcly accessing or closing over context from construction.

For example:

```
// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: ResolversWithContext.ts
// @include: ResolversWithContext

// @filename: QueriesFromLayers.ts
// ---cut---
import { Effect, Context, Layer, RequestResolver } from "effect"
import * as API from "./API"
import * as Model from "./Model"
import * as Requests from "./Requests"
import * as ResolversWithContext from "./ResolversWithContext"

export class TodosService extends Context.Tag("TodosService")<
  TodosService,
  {
    getTodos: Effect.Effect<Array<Model.Todo>, Model.GetTodosError>
  }
>() {}

export const TodosServiceLive = Layer.effect(
  TodosService,
  Effect.gen(function* () {
    const http = yield* ResolversWithContext.HttpService
    const resolver = RequestResolver.fromEffect(
      (request: Requests.GetTodos) =>
        Effect.tryPromise<Array<Model.Todo>, Model.GetTodosError>({
          try: () =>
            http
              .fetch("https://api.example.demo/todos")
              .then((res) => res.json()),
          catch: () => new Model.GetTodosError()
        })
    )
    return {
      getTodos: Effect.request(Requests.GetTodos({}), resolver)
    }
  })
)

export const getTodos: Effect.Effect<
  Array<Model.Todo>,
  Model.GetTodosError,
  TodosService
> = Effect.andThen(TodosService, (service) => service.getTodos)
```

This way is probably the best for most of the cases given that layers are the natural primitive where to wire services together.

Caching

While we have significantly optimized request batching, there's another area that can enhance our application's efficiency: caching. Without caching, even with optimized batch processing, the same requests could be executed multiple times, leading to unnecessary data fetching.

In the Effect library, caching is handled through built-in utilities that allow requests to be stored temporarily, preventing the need to re-fetch data that hasn't changed. This feature is crucial for reducing the load on both the server and the network, especially in applications that make frequent

similar requests.

Here's how you can implement caching for the `getUserById` query:

```
// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: Resolvers.ts
// @include: Resolvers

// @filename: Queries.ts
// ---cut---
import { Effect } from "effect"
import * as Requests from "./Requests"
import * as Resolvers from "./Resolvers"

export const getUserById = (id: number) =>
  Effect.request(
    Requests.GetUserById({ id }),
    Resolvers.GetUserByIdResolver
  ).pipe(Effect.withRequestCaching(true))
```

Final Program

Assuming you've wired everything up correctly:

```
// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: Resolvers.ts
// @include: Resolvers

// @filename: Queries.ts
// @include: Queries

// @filename: index.ts
// ---cut---
import { Effect, Schedule } from "effect"
import * as Queries from "./Queries"

const program = Effect.gen(function* () {
  const todos = yield* Queries.getTodos
  yield* Effect.forEach(todos, (todo) => Queries.notifyOwner(todo), {
    concurrency: "unbounded"
  })
}).pipe(Effect.repeat(Schedule.fixed("10 seconds")))
```

With this program, the `getTodos` operation retrieves the todos for each user. Then, the `Effect.forEach` function is used to notify the owner of each todo concurrently, without waiting for the notifications to complete.

The `repeat` function is applied to the entire chain of operations, and it ensures that the program repeats every 10 seconds using a fixed schedule. This means that the entire process, including fetching todos and sending notifications, will be executed repeatedly with a 10-second interval.

The program incorporates a caching mechanism, which prevents the same `GetUserById` operation from being executed more than once within a span of 1 minute. This default caching behavior helps optimize the program's execution and reduces unnecessary requests to fetch user data.

Furthermore, the program is designed to send emails in batches, allowing for efficient processing and better utilization of resources.

Customizing Request Caching

In real-world applications, effective caching strategies can significantly improve performance by reducing redundant data fetching. The Effect library provides flexible caching mechanisms that can be tailored for specific parts of your application or applied globally.

There may be scenarios where different parts of your application have unique caching requirements—some might benefit from a localized cache, while others might need a global cache setup. Let’s explore how you can configure a custom cache to meet these varied needs.

Creating a Custom Cache

Here's how you can create a custom cache and apply it to part of your application. This example demonstrates setting up a cache that repeats a task every 10 seconds, caching requests with specific parameters like capacity and TTL (time-to-live).

```
// @filename: Model.ts
// @include: Model

// @filename: API.ts
// @include: API

// @filename: Requests.ts
// @include: Requests

// @filename: Resolvers.ts
// @include: Resolvers

// @filename: Queries.ts
// @include: Queries

// @filename: index.ts
// ---cut---
import { Effect, Schedule, Layer, Request } from "effect"
import * as Queries from "./Queries"

const program = Effect.gen(function* () {
  const todos = yield* Queries.getTodos
  yield* Effect.forEach(todos, (todo) => Queries.notifyOwner(todo), {
    concurrency: "unbounded"
  })
}).pipe(
  Effect.repeat(Schedule.fixed("10 seconds")),
  Effect.provide(
    Layer.setRequestCache(
      Request.makeCache({ capacity: 256, timeToLive: "60 minutes" })
    )
  )
)
```

Direct Cache Application

You can also construct a cache using `Request.makeCache` and apply it directly to a specific program using `Effect.withRequestCache`. This method ensures that all requests originating from the specified program are managed through the custom cache, provided that caching is enabled.

Ref

Location: 400-guides/600-state-management/100-ref

Learn how to leverage Effect's `Ref` data type for efficient state management in your programs. Understand the importance of managing state in dynamic applications and the challenges posed by traditional approaches. Dive into the powerful capabilities of `Ref`, a mutable reference that provides a controlled way to handle mutable state and ensure safe updates in a concurrent environment. Explore practical examples, from simple counters to complex scenarios involving shared state and concurrent interactions. Enhance your programming skills by mastering the effective use of `Ref` for state management in your Effect programs.

When we write programs, it is common to need to keep track of some form of state during the execution of the program. State refers to any data that can change as the program runs. For example, in a counter application, the count value changes as the user increments or decrements it. Similarly, in a banking application, the account balance changes as deposits and withdrawals are made. State management is crucial to building interactive and dynamic applications.

In traditional imperative programming, one common way to store state is using variables. However, this approach can introduce bugs, especially when the state is shared between multiple components or functions. As the program becomes more complex, managing shared state can become challenging.

To overcome these issues, Effect introduces a powerful data type called `Ref`, which represents a mutable reference. With `Ref`, we can share state between different parts of our program without relying on mutable variables directly. Instead, `Ref` provides a controlled way to handle mutable state and safely update it in a concurrent environment.

Effect's `Ref` data type enables communication between different fibers in your program. This capability is crucial in concurrent programming, where multiple tasks may need to access and update shared state simultaneously.

In this guide, we will explore how to use the `Ref` data type to manage state in your programs effectively. We will cover simple examples like counting, as well as more complex scenarios where state is shared between different parts of the program. Additionally, we will show how to use `Ref` in a concurrent environment, allowing multiple tasks to interact with shared state safely.

Let's dive in and see how we can leverage `Ref` for effective state management in your Effect programs.

Using Ref

Let's explore how to use the `Ref` data type with a simple example of a counter:

```
import { Effect, Ref } from "effect"

export class Counter {
  inc: Effect.Effect<void>
  dec: Effect.Effect<void>
```

```
  get: Effect.Effect<number>

  constructor(private value: Ref.Ref<number>) {
    this.inc = Ref.update(this.value, (n) => n + 1)
    this.dec = Ref.update(this.value, (n) => n - 1)
    this.get = Ref.get(this.value)
  }
}

export const make = Effect.andThen(Ref.make(0), (value) => new Counter(value))

// @include: Counter
```

Here is the usage example of the `Counter`:

<Info> All the operations on the `Ref` data type are effectful. So when we are reading from or writing to a `Ref`, we are performing an effectful operation. </Info>

Using Ref in a Concurrent Environment

We can use this counter in a concurrent environment, such as counting the number of requests in a RESTful API. For this example, let's update the counter concurrently:

Using Ref as a Service

You can also pass a `Ref` as a service to share state between different parts of your program. Let's see how this works:

Note that we use `Effect.provideServiceEffect` instead of `Effect.provideService` to provide an actual implementation of the `MyState` service because all the operations on the `Ref` data type are effectful, including the creation `Ref.make(0)`.

Sharing state between Fibers

Let's consider an example where we want to read names from user input until the user enters the command `"q"` to exit.

First, let's introduce a `readLine` utility to read user input (ensure you have `@types/node` installed):

```
// @types: node
import { Effect } from "effect"
import * as NodeReadLine from "node:readline"

export const readLine = (
  message: string
): Effect.Effect<string> =>
  Effect.promise(
    () =>
      new Promise((resolve) => {
        const rl = NodeReadLine.createInterface({
          input: process.stdin,
          output: process.stdout
        })
        rl.question(message, (answer) => {
          rl.close()
          resolve(answer)
        })
      })
  )

// @include: ReadLine
```

Now, let's take a look at the main program:

Now that we have learned how to use the `Ref` data type, we can use it to manage the state concurrently. For example, assume while we are reading from the console, we have another fiber that is trying to update the state from a different source:

State Management

Location: 400-guides/600-state-management/index

State Management

SynchronizedRef

Location: 400-guides/600-state-management/200-synchronizedref

Discover the power of `SynchronizedRef` in Effect, a mutable reference enabling the atomic and effectful update of shared state. Building on the foundation of `Ref`, `SynchronizedRef` introduces the unique `updateEffect` function, allowing execution of effectful operations to modify the shared state. Dive into practical examples demonstrating the distinct capabilities of `SynchronizedRef`, such as parallel updates with consistent sequencing

and real-world scenarios involving API requests and state updates. Elevate your understanding of concurrent state management with this advanced feature in Effect programming.

`SynchronizedRef<A>` serves as a **mutable reference** to a value of type `A`. With it, we can store **immutable** data and perform updates **atomically and effectfully**.

<Info> Most of the operations for `SynchronizedRef` are similar to those of `Ref`. If you're not already familiar with `Ref`, it's recommended to read about the Ref concept first. </Info>

The distinctive function in `SynchronizedRef` is `updateEffect`. This function takes an **effectful operation** and executes it to modify the shared state. This is the key feature setting `SynchronizedRef` apart from `Ref`.

```
import { Effect, SynchronizedRef } from "effect"

const program = Effect.gen(function* () {
  const ref = yield* SynchronizedRef.make("current")
  // Simulating an effectful update operation
  const updateEffect = Effect.succeed("update")
  yield* SynchronizedRef.updateEffect(ref, () => updateEffect)
  const value = yield* SynchronizedRef.get(ref)
  return value
})

Effect.runPromise(program).then(console.log)
/*
Output:
update
*/
```

In real-world applications, there are scenarios where we need to execute an effect (e.g., querying a database) and then update the shared state accordingly. This is where `SynchronizedRef` shines, allowing us to update shared state in an actor-model fashion. We have a shared mutable state, but for each distinct command or message, we want to execute our effect and update the state.

We can pass an effectful program into every single update. All of these updates will be performed in parallel, but the results will be sequenced in such a way that they only affect the state at different times, resulting in a consistent state at the end.

In the following example, we send a `getAge` request for each user, updating the state accordingly:

```
import { Effect, SynchronizedRef } from "effect"

// Simulate API
const getAge = (userId: number) =>
  Effect.succeed({ userId, age: userId * 10 })

const users = [1, 2, 3, 4]

const meanAge = Effect.gen(function* () {
  const ref = yield* SynchronizedRef.make(0)

  const log = <R, E, A>(label: string, effect: Effect.Effect<A, E, R>) =>
    Effect.gen(function* () {
      const value = yield* SynchronizedRef.get(ref)
      yield* Effect.log(`${label} get: ${value}`)
      return yield* effect
    })

  const task = (id: number) =>
    log(
      `task ${id}`,
      SynchronizedRef.updateEffect(ref, (sumOfAges) =>
        Effect.gen(function* () {
          const user = yield* getAge(id)
          return sumOfAges + user.age
        })
      )
    )

  yield* task(1).pipe(
    Effect.zip(task(2), { concurrent: true }),
    Effect.zip(task(3), { concurrent: true }),
    Effect.zip(task(4), { concurrent: true })
  )

  const value = yield* SynchronizedRef.get(ref)
  return value / users.length
})

Effect.runPromise(meanAge).then(console.log)
/*
Output:
... fiber=#1 message="task 4 get: 0"
... fiber=#2 message="task 3 get: 40"
... fiber=#3 message="task 1 get: 70"
... fiber=#4 message="task 2 get: 80"
25
*/
```

Repetition

Location: 400-guides/550-scheduling/200-repetition

Discover the significance of repetition in effect-driven software development with functions like `repeat` and `repeatOrElse`. Explore repeat policies, which enable you to execute effects multiple times according to specific criteria. Learn the syntax and examples of `repeat` and `repeatOrElse` for effective handling of repeated actions, including optional fallback strategies for errors.

Repetition is a common requirement when working with effects in software development. It allows us to perform an effect multiple times according to a specific repetition policy.

repeat

The `repeat` function returns a new effect that repeats the given effect according to a specified schedule or until the first failure. The scheduled recurrences are in addition to the initial execution, so `Effect.repeat(action, Schedule.once)` executes `action` once initially, and if it succeeds, repeats it an additional time.

Success Example

```
import { Effect, Schedule, Console } from "effect"

const action = Console.log("success")

const policy = Schedule.addDelay(Schedule.recurs(2), () => "100 millis")

const program = Effect.repeat(action, policy)

Effect.runPromise(program).then((n) => console.log(`repetitions: ${n}`))
/*
Output:
success
success
success
repetitions: 2
*/
```

Failure Example

```
import { Effect, Schedule } from "effect"

let count = 0

// Define an async effect that simulates an action with possible failures
const action = Effect.async<string, string>((resume) => {
  if (count > 1) {
    console.log("failure")
    resume(Effect.fail("Uh oh!"))
  } else {
    count++
    console.log("success")
    resume(Effect.succeed("yay!"))
  }
})

const policy = Schedule.addDelay(Schedule.recurs(2), () => "100 millis")

const program = Effect.repeat(action, policy)

Effect.runPromiseExit(program).then(console.log)
/*
Output:
success
success
failure
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: 'Uh oh!' }
}
*/
```

Skipping the First Occurrence

If you want to skip the first occurrence of a repeat, you can use `Effect.schedule`:

```
import { Effect, Schedule, Console } from "effect"

const action = Console.log("success")

const policy = Schedule.addDelay(Schedule.recurs(2), () => "100 millis")

const program = Effect.schedule(action, policy)

Effect.runPromise(program).then((n) => console.log(`repetitions: ${n}`))
```

```
/*
Output:
success
success
repetitions: 2
*/
```

repeatN

The `repeatN` function returns a new effect that repeats the specified effect a given number of times or until the first failure. The repeats are in addition to the initial execution, so `Effect.repeatN(action, 1)` executes `action` once initially and then repeats it one additional time if it succeeds.

```
import { Effect, Console } from "effect"

const action = Console.log("success")

const program = Effect.repeatN(action, 2)

Effect.runPromise(program)
/*
Output:
success
success
success
*/
```

repeatOrElse

The `repeatOrElse` function returns a new effect that repeats the specified effect according to the given schedule or until the first failure. When a failure occurs, the failure value and schedule output are passed to a specified handler. Scheduled recurrences are in addition to the initial execution, so `Effect.repeat(action, Schedule.once)` executes `action` once initially and then repeats it an additional time if it succeeds.

```
import { Effect, Schedule } from "effect"

let count = 0

// Define an async effect that simulates an action with possible failures
const action = Effect.async<string, string>((resume) => {
  if (count > 1) {
    console.log("failure")
    resume(Effect.fail("Uh oh!"))
  } else {
    count++
    console.log("success")
    resume(Effect.succeed("yay!"))
  }
})

const policy = Schedule.addDelay(
  Schedule.recurs(2), // Repeat for a maximum of 2 times
  () => "100 millis" // Add a delay of 100 milliseconds between repetitions
)

const program = Effect.repeatOrElse(action, policy, () =>
  Effect.sync(() => {
    console.log("orElse")
    return count - 1
  })
)

Effect.runPromise(program).then((n) => console.log(`repetitions: ${n}`))
/*
Output:
success
success
failure
orElse
repetitions: 1
*/
```

Scheduling

Location: 400-guides/550-scheduling/index

Scheduling

Schedule Combinators

Location: 400-guides/550-scheduling/400-schedule-combinators

Explore the power of combining schedules in Effect to create sophisticated recurring patterns. Learn about key combinators such as `Union`, `Intersection`, and `Sequencing`. Witness the impact of `Jittering` on scheduling by introducing randomness to delays. Understand how to `Filter` inputs or outputs and modify delays with precision. Leverage `Tapping` to effectfully process each schedule input/output, providing insights into the execution flow. Elevate your understanding of schedules for efficient and flexible handling of effectful operations.

Schedules define stateful, possibly effectful, recurring schedules of events, and compose in a variety of ways. Combinators allow us to take schedules and combine them together to get other schedules.

To demonstrate the functionality of different schedules, we will be working with the following helper:

```
import { Effect, Schedule, TestClock, Fiber, TestContext } from "effect"

let start = 0
let i = 0

export const log = <A, Out>(
  action: Effect.Effect<A>,
  schedule: Schedule.Schedule<Out, void>
) => {
  Effect.gen(function* () {
    const fiber: Fiber.RuntimeFiber<[Out, number]> = yield* Effect.gen(
      function* () {
        yield* action
        const now = yield* TestClock.currentTimeMillis
        console.log(
          i === 0
            ? `delay: ${now - start}`
            : i === 10
              ? "..."
              : `#${i} delay: ${now - start}`
        )
        i++
        start = now
      }
    ).pipe(
      Effect.repeat(schedule.pipe(Schedule.intersect(Schedule.recurs(10)))),
      Effect.fork
    )
    yield* TestClock.adjust(Infinity)
    yield* Fiber.join(fiber)
  }).pipe(Effect.provide(TestContext.TestContext), Effect.runPromise)
}

declare const log: <A, Out>(
  action: Effect.Effect<A>,
  schedule: Schedule.Schedule<Out, void>
) => void
```


 <details> <summary>Click to see the implementation</summary>

```
// @include: Delay
```

</details>

The `log` helper logs the time delay between each execution. We will use this helper to showcase the behavior of various built-in schedules.

<Warning> The `log` helper accelerates time using TestClock, which means it simulates the passing of time that would normally occur in a real-world application. </Warning>

Composition

Schedules compose in the following primary ways:

- **Union**. This performs the union of the intervals of two schedules.
- **Intersection**. This performs the intersection of the intervals of two schedules.
- **Sequencing**. This concatenates the intervals of one schedule onto another.

Union

Combines two schedules through union, by recurring if either schedule wants to recur, using the minimum of the two delays between recurrences.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.union(
  Schedule.exponential("100 millis"),
  Schedule.spaced("1 second")
)
const action = Effect.void
```

```
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 100  < exponential
#2 delay: 200
#3 delay: 400
#4 delay: 800
#5 delay: 1000 < spaced
#6 delay: 1000
#7 delay: 1000
#8 delay: 1000
#9 delay: 1000
...
*/
```

When we use the combined schedule with `Effect.repeat`, we observe that the effect is executed repeatedly based on the minimum delay between the two schedules. In this case, the delay alternates between the exponential schedule (increasing delay) and the spaced schedule (constant delay).

Intersection

Combines two schedules through the intersection, by recurring only if both schedules want to recur, using the maximum of the two delays between recurrences.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.intersect(
  Schedule.exponential("10 millis"),
  Schedule.recurs(5)
)
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 10  < exponential
#2 delay: 20
#3 delay: 40
#4 delay: 80
#5 delay: 160
(end)         < recurs
*/
```

When we use the combined schedule with `Effect.repeat`, we observe that the effect is executed repeatedly only if both schedules want it to recur. The delay between recurrences is determined by the maximum delay between the two schedules. In this case, the delay follows the progression of the exponential schedule until the maximum number of recurrences specified by the recursive schedule is reached.

Sequencing

Combines two schedules sequentially, by following the first policy until it ends, and then following the second policy.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.andThen(
  Schedule.recurs(5),
  Schedule.spaced("1 second")
)
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 0    < recurs
#2 delay: 0
#3 delay: 0
#4 delay: 0
#5 delay: 0
#6 delay: 1000 < spaced
#7 delay: 1000
#8 delay: 1000
#9 delay: 1000
...
*/
```

When we use the combined schedule with `Effect.repeat`, we observe that the effect follows the policy of the first schedule (recurs) until it completes the specified number of recurrences. After that, it switches to the policy of the second schedule (spaced) and continues repeating the effect with the fixed delay between recurrences.

Jittering

A `jittered` is a combinator that takes one schedule and returns another schedule of the same type except for the delay which is applied randomly

When a resource is out of service due to overload or contention, retrying and backing off doesn't help us. If all failed API calls are backed off to the same point of time, they cause another overload or contention. Jitter adds some amount of randomness to the delay of the schedule. This helps us to avoid ending up accidentally synchronizing and taking the service down by accident.

Research shows that `Schedule.jittered(0.0, 1.0)` is very suitable for retrying.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.jittered(Schedule.exponential("10 millis"))
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 9.006765
#2 delay: 20.549507999999996
#3 delay: 45.86659000000001
#4 delay: 77.055037
#5 delay: 178.06722299999998
#6 delay: 376.056965
#7 delay: 728.732785
#8 delay: 1178.174953
#9 delay: 2331.4659370000004
...
*/
```

In this example, we use the `jittered` combinator to apply jitter to an exponential schedule. The exponential schedule increases the delay between each repetition exponentially. By adding jitter to the schedule, the delays become randomly adjusted within a certain range.

Filtering

We can filter inputs or outputs of a schedule with `whileInput` and `whileOutput`.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.whileOutput(Schedule.recurs(5), (n) => n <= 2)
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 0 < recurs
#2 delay: 0
#3 delay: 0
(end)       < whileOutput
*/
```

In this example, we create a schedule using `Schedule.recurs(5)` to repeat a certain action up to 5 times. However, we apply the `whileOutput` combinator with a predicate that filters out outputs greater than 2. As a result, the schedule stops producing outputs once the value exceeds 2, and the repetition ends.

Modifying

Modifies the delay of a schedule.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"
```

```
const schedule = Schedule.modifyDelay(
  Schedule.spaced("1 second"),
  (_) => "100 millis"
)
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 100 < modifyDelay
#2 delay: 100
#3 delay: 100
#4 delay: 100
#5 delay: 100
#6 delay: 100
#7 delay: 100
#8 delay: 100
#9 delay: 100
...
*/
```

Tapping

Whenever we need to effectfully process each schedule input/output, we can use `tapInput` and `tapOutput`.

We can use these two functions for logging purposes:

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect, Console } from "effect"
import { log } from "./Delay"

const schedule = Schedule.tapOutput(Schedule.recurs(2), (n) =>
  Console.log(`repeating ${n}`)
)
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
repeating 0
#1 delay: 0
repeating 1
#2 delay: 0
repeating 2
*/
```

Examples

Location: 400-guides/550-scheduling/500-examples

Practical Examples of Using Schedule

Making API Calls and Handling Timeouts

For our API calls to third-party services, we have a few requirements. We want to ensure that if the entire function takes more than `4` seconds to execute, it's interrupted. Additionally, we'll set up the system to retry the API call a maximum of `2` times.

Solution

```
import { NodeRuntime } from "@effect/platform-node"
import { Console, Effect } from "effect"

const getJson = (url: string) =>
  Effect.tryPromise(() =>
    fetch(url).then((res) => {
      if (!res.ok) {
        console.log("error")
        throw new Error(res.statusText)
      }
      console.log("ok")
      return res.json() as unknown
    })
  )

const program = (url: string) =>
  getJson(url).pipe(
    Effect.retry({ times: 2 }),
    Effect.timeout("4 seconds"),
    Effect.catchAll(Console.error)
  )
```

```
// testing the happy path
NodeRuntime.runMain(program("https://dummyjson.com/products/1?delay=1000"))
/*
Output:
ok
*/

// testing the timeout
// NodeRuntime.runMain(program("https://dummyjson.com/products/1?delay=5000"))
/*
Output:
TimeoutException
*/

// testing API errors
// NodeRuntime.runMain(
//   program("https://dummyjson.com/auth/products/1?delay=500")
// )
/*
Output:
error
error
error
UnknownException: Forbidden
*/
```

Implementing Conditional Retries

We want to implement a mechanism to retry an API call only if it encounters certain types of errors.

Solution

```
import { NodeRuntime } from "@effect/platform-node"
import { Console, Effect } from "effect"

class Err extends Error {
  constructor(
    message: string,
    readonly status: number
  ) {
    super(message)
  }
}

const getJson = (url: string) =>
  Effect.tryPromise({
    try: () =>
      fetch(url).then((res) => {
        if (!res.ok) {
          console.log(res.status)
          throw new Err(res.statusText, res.status)
        }
        return res.json() as unknown
      }),
    catch: (e) => e as Err
  })

const program = (url: string) =>
  getJson(url).pipe(
    // Retry if the error is a 403
    Effect.retry({ while: (err) => err.status === 403 }),
    Effect.catchAll(Console.error)
  )

// testing 403
NodeRuntime.runMain(
  program("https://dummyjson.com/auth/products/1?delay=1000")
)
/*
Output:
403
403
403
403
...
*/

// testing 404
// NodeRuntime.runMain(program("https://dummyjson.com/-"))
/*
Output:
404
Err [Error]: Not Found
*/
```

Running Scheduled Effects Until Completion

We can use schedules to run an effect periodically until another long-running effect completes. This can be useful for tasks like polling or periodic logging.

Solution

```
import { Effect, Console, Schedule } from "effect"

const longRunningEffect = Console.log("done").pipe(Effect.delay("5 seconds"))

const action = Console.log("action...")

const schedule = Schedule.fixed("1.5 seconds")

const program = Effect.race(
  Effect.repeat(action, schedule),
  longRunningEffect
)

Effect.runPromise(program)
/*
Output:
action...
action...
action...
action...
done
*/
```

Built-in Schedules

Location: 400-guides/550-scheduling/300-built-in-schedules

Unlock the power of scheduling in Effect with built-in schedules. Dive into various schedules like `forever`, `once`, and `recurs`, each offering unique recurrence patterns. Witness the behavior of `spaced` and `fixed` schedules, understanding how they space repetitions at specific intervals. Delve into advanced schedules like `exponential` and `fibonacci`, providing controlled recurrence with increasing delays. Master the art of scheduling for precise and efficient execution of effectful operations.

To demonstrate the functionality of different schedules, we will be working with the following helper:

```
import { Effect, Schedule, TestClock, Fiber, TestContext } from "effect"

let start = 0
let i = 0

export const log = <A, Out>(
  action: Effect.Effect<A>,
  schedule: Schedule.Schedule<Out, void>
) => {
  Effect.gen(function* () {
    const fiber: Fiber.RuntimeFiber<[Out, number]> = yield* Effect.gen(
      function* () {
        yield* action
        const now = yield* TestClock.currentTimeMillis
        console.log(
          i === 0
            ? `delay: ${now - start}`
            : i === 10
              ? "..."
              : `#${i} delay: ${now - start}`
        )
        i++
        start = now
      }
    ).pipe(
      Effect.repeat(schedule.pipe(Schedule.intersect(Schedule.recurs(10)))),
      Effect.fork
    )
    yield* TestClock.adjust(Infinity)
    yield* Fiber.join(fiber)
  }).pipe(Effect.provide(TestContext.TestContext), Effect.runPromise)
}

declare const log: <A, Out>(
  action: Effect.Effect<A>,
  schedule: Schedule.Schedule<Out, void>
) => void
```


 <details> <summary>Click to see the implementation</summary>

```
// @include: Delay
```

</details>

The `log` helper logs the time delay between each execution. We will use this helper to showcase the behavior of various built-in schedules.

<Warning> The `log` helper accelerates time using TestClock, which means it simulates the passing of time that would normally occur in a real-world application. </Warning>

forever

A schedule that always recurs and produces number of recurrence at each run.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.forever
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 0 < forever
#2 delay: 0
#3 delay: 0
#4 delay: 0
#5 delay: 0
#6 delay: 0
#7 delay: 0
#8 delay: 0
#9 delay: 0
...
*/
```

once

A schedule that recurs one time.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.once
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 0 < once
*/
```

recurs

A schedule that only recurs the specified number of times.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.recurs(5)
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 0 < recurs
#2 delay: 0
#3 delay: 0
#4 delay: 0
#5 delay: 0
*/
```

Recurring at specific intervals

In the context of scheduling, two commonly used schedules are `spaced` and `fixed`. While they both involve recurring at specific intervals, they have a fundamental difference in how they determine the timing of repetitions.

The `spaced` schedule creates a recurring pattern where each repetition is spaced apart by a specified duration. This means that there is a delay between the completion of one repetition and the start of the next. The duration between repetitions remains constant, providing a consistent spacing pattern.

On the other hand, the `fixed` schedule recurs on a fixed interval, regardless of the duration of the actions or the completion time of previous repetitions. It operates independently of the execution time, ensuring a regular recurrence at the specified interval.

spaced

A schedule that recurs continuously, each repetition spaced the specified duration from the last run.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.spaced("200 millis")
const action = Effect.delay(Effect.void, "100 millis")
log(action, schedule)
/*
Output:
delay: 100
#1 delay: 300 < spaced
#2 delay: 300
#3 delay: 300
#4 delay: 300
#5 delay: 300
#6 delay: 300
#7 delay: 300
#8 delay: 300
#9 delay: 300
...
*/
```

The first delay is approximately 100 milliseconds, as the initial execution is not affected by the schedule. Subsequent delays are approximately 200 milliseconds apart, demonstrating the effect of the `spaced` schedule.

fixed

A schedule that recurs on a fixed interval. Returns the number of repetitions of the schedule so far.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.fixed("200 millis")
const action = Effect.delay(Effect.void, "100 millis")
log(action, schedule)
/*
Output:
delay: 100
#1 delay: 300 < fixed
#2 delay: 200
#3 delay: 200
#4 delay: 200
#5 delay: 200
#6 delay: 200
#7 delay: 200
#8 delay: 200
#9 delay: 200
...
*/
```

The first delay is approximately 100 milliseconds, as the initial execution is not affected by the schedule. Subsequent delays are consistently around 200 milliseconds apart, demonstrating the effect of the `fixed` schedule.

exponential

A schedule that recurs using exponential backoff

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"
```

```
const schedule = Schedule.exponential("10 millis")
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 10 < exponential
#2 delay: 20
#3 delay: 40
#4 delay: 80
#5 delay: 160
#6 delay: 320
#7 delay: 640
#8 delay: 1280
#9 delay: 2560
...
*/
```

fibonacci

A schedule that always recurs, increasing delays by summing the preceding two delays (similar to the fibonacci sequence). Returns the current duration between recurrences.

```
// @filename: Delay.ts
// @include: Delay

// @filename: index.ts
// ---cut---
import { Schedule, Effect } from "effect"
import { log } from "./Delay"

const schedule = Schedule.fibonacci("10 millis")
const action = Effect.void
log(action, schedule)
/*
Output:
delay: 0
#1 delay: 10 < fibonacci
#2 delay: 10
#3 delay: 20
#4 delay: 30
#5 delay: 50
#6 delay: 80
#7 delay: 130
#8 delay: 210
#9 delay: 340
...
*/
```

Introduction to Scheduling

Location: 400-guides/550-scheduling/100-introduction

Explore the fundamental concepts of scheduling in Effect using `Schedule`. These immutable values define recurring patterns for executing effects based on input values and internal state. Learn about the composability of schedules, allowing you to create sophisticated recurrence patterns by combining and modifying existing schedules.

Scheduling is an important concept in Effect that allows you to define recurring effectful operations. It involves the use of `Schedule<Out, In, Context>`, which is an **immutable value** that describes a scheduled pattern for executing effects.

A `Schedule` operates by consuming values of type `In` (such as errors in the case of `retry`, or values in the case of `repeat`) and producing values of type `Out`. It determines when to halt or continue the execution based on input values and its internal state.

The inclusion of a `Context` parameter allows the schedule to leverage additional services or resources as needed.

Schedules are defined as a collection of intervals spread out over time. Each interval represents a window during which the recurrence of an effect is possible.

Retrying and Repetition

In the realm of scheduling, there are two related concepts: Retrying and Repetition. While they share the same underlying idea, they differ in their focus. Retrying aims to handle failures by executing an effect again, while repetition focuses on executing an effect repeatedly to achieve a desired outcome.

When using schedules for retrying or repetition, each interval's starting boundary determines when the effect will be executed again. For example, in retrying, if an error occurs, the schedule defines when the effect should be retried.

Composability of Schedules

Schedules are composable, meaning you can combine simple schedules to create more complex recurrence patterns. Operators like `union` or `intersect` allow you to build sophisticated schedules by combining and modifying existing ones. This flexibility enables you to tailor the scheduling behavior to meet specific requirements.

Default Services

Location: 400-guides/200-context-management/200-default-services

Explore the default services in Effect - Clock, Console, Random, ConfigProvider, and Tracer. Learn how Effect automatically provides live versions of these services, eliminating the need for explicit implementations in your programs.

Effect comes equipped with five pre-built services:

```
type DefaultServices = Clock | Console | Random | ConfigProvider | Tracer
```

When we employ these services, there's no need to explicitly provide their implementations. Effect automatically supplies live versions of these services to our effects, sparing us from manual setup.

```
import { Effect, Clock, Console } from "effect"

const program = Effect.gen(function* () {
  const now = yield* Clock.currentTimeMillis
  yield* Console.log(`Application started at ${new Date(now)}`)
})
```

As you can observe, even if our program utilizes both `Clock` and `Console`, the `Requirements` parameter, representing the services required for the effect to execute, remains set to `never`. Effect takes care of handling these services seamlessly for us.

Requirements Management

Location: 400-guides/200-context-management/index

Requirements Management

Managing Layers

Location: 400-guides/200-context-management/300-layers

Learn how to use Layers to control the construction of service dependencies and manage the "dependency graph" of your program more effectively.

In the previous sections, you learned how to create effects which depend on some service to be provided in order to execute, as well as how to provide that service to an Effect.

However, what if we have a service within our effect program that has dependencies on other services in order to be built? We want to avoid leaking these implementation details into the service interface.

To represent the "dependency graph" of our program and manage these dependencies more effectively, we can utilize a powerful abstraction called **Layer**.

Layers act as **constructors** for creating services, allowing us to manage dependencies during construction rather than at the service level. This approach helps to keep our service interfaces clean and focused.

Concept	Description
Service	A reusable component providing specific functionality, used across different parts of an application.
Tag	A unique identifier representing a **service**, allowing Effect to locate and use it.
Context	A collection storing services, functioning like a map with **tags** as keys and **services** as values.
Layer	An abstraction for constructing **services**, managing dependencies during construction rather than at the service level.

In this guide, we will cover the following topics:

- Using Layers to control the construction of services.
- Building a dependency graph with Layers.
- Providing a Layer to an effect.

Designing the Dependency Graph

Let's imagine that we are building a web application! We could imagine that the dependency graph for an application where we need to manage configuration, logging, and database access might look something like this:

- The `Config` service provides application configuration.
- The `Logger` service depends on the `Config` service.
- The `Database` service depends on both the `Config` and `Logger` services.

Our goal is to build the `Database` service along with its direct and indirect dependencies. This means we need to ensure that the `Config` service is available for both `Logger` and `Database`, and then provide these dependencies to the `Database` service.

Now let's take our dependency graph and translate it into code.

Creating Layers

We will use Layers to construct the `Database` service instead of providing a service implementation directly as we did in the Managing Services guide. Layers are a way of separating implementation details from the service itself.

<Idea> When a service has its own requirements, it's best to separate implementation details into layers. Layers act as **constructors** for creating the service, allowing us to handle dependencies at the construction level rather than the service level. </Idea>

A `Layer<RequirementsOut, Error, RequirementsIn>` represents a blueprint for constructing a `RequirementsOut`. It takes a value of type `RequirementsIn` as input and may potentially produce an error of type `Error` during the construction process.

In our case, the `RequirementsOut` type represents the service we want to construct, while `RequirementsIn` represents the dependencies required for construction.

<Info> For simplicity, let's assume that we won't encounter any errors during the value construction (meaning `Error = never`). </Info>

Now, let's determine how many layers we need to implement our dependency graph:

Layer	Dependencies	Type
`ConfigLive`	The `Config` service does not depend on any other services	`Layer<Config>`
`LoggerLive`	The `Logger` service depends on the `Config` service	`Layer<Logger, never, Config>`
`DatabaseLive`	The `Database` service depends on `Config` and `Logger`	`Layer<Database, never, Config | Logger>`

<Idea> A common convention when naming the `Layer` for a particular service is to add a `Live` suffix for the "live" implementation and a `Test` suffix for the "test" implementation. For example, for a `Database` service, the `DatabaseLive` would be the layer you provide in your application and the `DatabaseTest` would be the layer you provide in your tests. </Idea>

When a service has multiple dependencies, they are represented as a **union type**. In our case, the `Database` service depends on both the `Config` and `Logger` services. Therefore, the type for the `DatabaseLive` layer will be `Layer<Database, never, Config | Logger>`.

Config

The `Config` service does not depend on any other services, so `ConfigLive` will be the simplest layer to implement. Just like in the Managing Services guide, we must create a `Tag` for the service. And because the service has no dependencies, we can create the layer directly using `Layer.succeed`:

```
import { Effect, Context, Layer } from "effect"

// Create a tag for the Config service
class Config extends Context.Tag("Config")<
  Config,
  {
    readonly getConfig: Effect.Effect<{
      readonly logLevel: string
      readonly connection: string
    }>
  }
>() {}

const ConfigLive = Layer.succeed(
  Config,
  Config.of({
    getConfig: Effect.succeed({
      logLevel: "INFO",
      connection: "mysql://username:password@hostname:port/database_name"
    })
  })
)
```

Looking at the type of `ConfigLive` we can observe:

- `RequirementsOut` is `Config`, indicating that constructing the layer will produce a `Config` service
- `Error` is `never`, indicating that layer construction cannot fail
- `RequirementsIn` is `never`, indicating that the layer has no dependencies

Note that, to construct `ConfigLive`, we used the `Config.of` constructor. However, this is merely a helper to ensure correct type inference for the implementation. It's possible to skip this helper and construct the implementation directly as a simple object:

```
import { Effect, Context, Layer } from "effect"

class Config extends Context.Tag("Config")<
  Config,
  {
    readonly getConfig: Effect.Effect<{
      readonly logLevel: string
```

```
      readonly connection: string
    }>
  }
>() {}

// ---cut---
const ConfigLive = Layer.succeed(Config, {
  getConfig: Effect.succeed({
    logLevel: "INFO",
    connection: "mysql://username:password@hostname:port/database_name"
  })
})
```

Logger

Now we can move on to the implementation of the `Logger` service, which depends on the `Config` service to retrieve some configuration.

Just like we did in the Managing Services guide, we can map over the `Config` tag to "extract" the service from the context.

Given that using the `Config` tag is an effectful operation, we use `Layer.effect` to create a `Layer` from the resulting `Effect`.

```
import { Effect, Context, Layer } from "effect"

class Config extends Context.Tag("Config")<
  Config,
  {
    readonly getConfig: Effect.Effect<{
      readonly logLevel: string
      readonly connection: string
    }>
  }
>() {}

// ---cut---
class Logger extends Context.Tag("Logger")<
  Logger,
  { readonly log: (message: string) => Effect.Effect<void> }
>() {}

const LoggerLive = Layer.effect(
  Logger,
  Effect.gen(function* () {
    const config = yield* Config
    return {
      log: (message) =>
        Effect.gen(function* () {
          const { logLevel } = yield* config.getConfig
          console.log(`[${logLevel}] ${message}`)
        })
    }
  })
)
```

Looking at the type of `LoggerLive` we can observe:

- `RequirementsOut` is `Logger`
- `Error` is `never`, indicating that layer construction cannot fail
- `RequirementsIn` is `Config`, indicating that the layer has a requirement

Database

Finally, we can use our `Config` and `Logger` services to implement the `Database` service.

```
import { Effect, Context, Layer } from "effect"

class Config extends Context.Tag("Config")<
  Config,
  {
    readonly getConfig: Effect.Effect<{
      readonly logLevel: string
      readonly connection: string
    }>
  }
>() {}

class Logger extends Context.Tag("Logger")<
  Logger,
  { readonly log: (message: string) => Effect.Effect<void> }
>() {}

// ---cut---
class Database extends Context.Tag("Database")<
  Database,
  { readonly query: (sql: string) => Effect.Effect<unknown> }
>() {}

const DatabaseLive = Layer.effect(
```

```
  Database,
  Effect.gen(function* () {
    const config = yield* Config
    const logger = yield* Logger
    return {
      query: (sql: string) =>
        Effect.gen(function* () {
          yield* logger.log(`Executing query: ${sql}`)
          const { connection } = yield* config.getConfig
          return { result: `Results from ${connection}` }
        })
    }
  })
)
```

Looking at the type of `DatabaseLive` we can observe that the `RequirementsIn` type is `Config | Logger`, i.e., the `Database` service requires both `Config` and `Logger` services.

Combining Layers

Layers can be combined in two primary ways: merging and composing.

Merging Layers

Layers can be combined through merging using the `Layer.merge` combinator:

```
Layer.merge(layer1, layer2)
```

When we merge two layers, the resulting layer:

- requires all the services that both of them require.
- produces all services that both of them produce.

For example, in our web application above, we can merge our `ConfigLive` and `LoggerLive` layers into a single `AppConfigLive` layer, which retains the requirements of both layers (`never | Config = Config`) and the outputs of both layers (`Config | Logger`):

```
import { Effect, Context, Layer } from "effect"

class Config extends Context.Tag("Config")<
  Config,
  {
    readonly getConfig: Effect.Effect<{
      readonly logLevel: string
      readonly connection: string
    }>
  }
>() {}

const ConfigLive = Layer.succeed(
  Config,
  Config.of({
    getConfig: Effect.succeed({
      logLevel: "INFO",
      connection: "mysql://username:password@hostname:port/database_name"
    })
  })
)

class Logger extends Context.Tag("Logger")<
  Logger,
  { readonly log: (message: string) => Effect.Effect<void> }
>() {}

const LoggerLive = Layer.effect(
  Logger,
  Effect.gen(function* () {
    const config = yield* Config
    return {
      log: (message) =>
        Effect.gen(function* () {
          const { logLevel } = yield* config.getConfig
          console.log(`[${logLevel}] ${message}`)
        })
    }
  })
)

// ---cut---
const AppConfigLive = Layer.merge(ConfigLive, LoggerLive)
```

Composing Layers

Layers can be composed using the `Layer.provide` function:

```
import { Layer } from "effect"

declare const inner: Layer.Layer<"OutInner", never, "InInner">
declare const outer: Layer.Layer<"InInner", never, "InOuter">

const composition = inner.pipe(Layer.provide(outer))
```

Sequential composition of layers implies that the output of one layer (`outer`) is supplied as the input for the inner layer (`inner`), resulting in a single layer with the requirements of the first layer and the output of the second.

Now we can compose the `AppConfigLive` layer with the `DatabaseLive` layer:

```
import { Effect, Context, Layer } from "effect"

class Config extends Context.Tag("Config")<
  Config,
  {
    readonly getConfig: Effect.Effect<{
      readonly logLevel: string
      readonly connection: string
    }>
  }
>() {}

const ConfigLive = Layer.succeed(
  Config,
  Config.of({
    getConfig: Effect.succeed({
      logLevel: "INFO",
      connection: "mysql://username:password@hostname:port/database_name"
    })
  })
)

class Logger extends Context.Tag("Logger")<
  Logger,
  { readonly log: (message: string) => Effect.Effect<void> }
>() {}

const LoggerLive = Layer.effect(
  Logger,
  Effect.gen(function* () {
    const config = yield* Config
    return {
      log: (message) =>
        Effect.gen(function* () {
          const { logLevel } = yield* config.getConfig
          console.log(`[${logLevel}] ${message}`)
        })
    }
  })
)

class Database extends Context.Tag("Database")<
  Database,
  { readonly query: (sql: string) => Effect.Effect<unknown> }
>() {}

const DatabaseLive = Layer.effect(
  Database,
  Effect.gen(function* () {
    const config = yield* Config
    const logger = yield* Logger
    return {
      query: (sql: string) =>
        Effect.gen(function* () {
          yield* logger.log(`Executing query: ${sql}`)
          const { connection } = yield* config.getConfig
          return { result: `Results from ${connection}` }
        })
    }
  })
)

// ---cut---
const AppConfigLive = Layer.merge(ConfigLive, LoggerLive)

const MainLive = DatabaseLive.pipe(
  // provides the config and logger to the database
  Layer.provide(AppConfigLive),
  // provides the config to AppConfigLive
  Layer.provide(ConfigLive)
)
```

Merging and Composing Layers

Let's say we want our `MainLive` layer to return both the `Config` and `Database` services. We can achieve this with `Layer.provideMerge`:

```
import { Effect, Context, Layer } from "effect"

class Config extends Context.Tag("Config")<
  Config,
  {
    readonly getConfig: Effect.Effect<{
      readonly logLevel: string
      readonly connection: string
    }>
  }
>() {}

const ConfigLive = Layer.succeed(
  Config,
  Config.of({
    getConfig: Effect.succeed({
      logLevel: "INFO",
      connection: "mysql://username:password@hostname:port/database_name"
    })
  })
)

class Logger extends Context.Tag("Logger")<
  Logger,
  { readonly log: (message: string) => Effect.Effect<void> }
>() {}

const LoggerLive = Layer.effect(
  Logger,
  Effect.gen(function* () {
    const config = yield* Config
    return {
      log: (message) =>
        Effect.gen(function* () {
          const { logLevel } = yield* config.getConfig
          console.log(`[${logLevel}] ${message}`)
        })
    }
  })
)

class Database extends Context.Tag("Database")<
  Database,
  { readonly query: (sql: string) => Effect.Effect<unknown> }
>() {}

const DatabaseLive = Layer.effect(
  Database,
  Effect.gen(function* () {
    const config = yield* Config
    const logger = yield* Logger
    return {
      query: (sql: string) =>
        Effect.gen(function* () {
          yield* logger.log(`Executing query: ${sql}`)
          const { connection } = yield* config.getConfig
          return { result: `Results from ${connection}` }
        })
    }
  })
)

// ---cut---
const AppConfigLive = Layer.merge(ConfigLive, LoggerLive)

const MainLive = DatabaseLive.pipe(
  Layer.provide(AppConfigLive),
  Layer.provideMerge(ConfigLive)
)
```

Providing a Layer to an Effect

Now that we have assembled the fully resolved `MainLive` for our application, we can provide it to our program to satisfy the program's requirements using `Effect.provide`:

```
import { Effect, Context, Layer } from "effect"

class Config extends Context.Tag("Config")<
  Config,
  {
    readonly getConfig: Effect.Effect<{
      readonly logLevel: string
      readonly connection: string
    }>
  }
>() {}

const ConfigLive = Layer.succeed(
  Config,
```

```
  Config.of({
    getConfig: Effect.succeed({
      logLevel: "INFO",
      connection: "mysql://username:password@hostname:port/database_name"
    })
  })
)

class Logger extends Context.Tag("Logger")<
  Logger,
  { readonly log: (message: string) => Effect.Effect<void> }
>() {}

const LoggerLive = Layer.effect(
  Logger,
  Effect.gen(function* () {
    const config = yield* Config
    return {
      log: (message) =>
        Effect.gen(function* () {
          const { logLevel } = yield* config.getConfig
          console.log(`[${logLevel}] ${message}`)
        })
    }
  })
)

class Database extends Context.Tag("Database")<
  Database,
  { readonly query: (sql: string) => Effect.Effect<unknown> }
>() {}

const DatabaseLive = Layer.effect(
  Database,
  Effect.gen(function* () {
    const config = yield* Config
    const logger = yield* Logger
    return {
      query: (sql: string) =>
        Effect.gen(function* () {
          yield* logger.log(`Executing query: ${sql}`)
          const { connection } = yield* config.getConfig
          return { result: `Results from ${connection}` }
        })
    }
  })
)

const AppConfigLive = Layer.merge(ConfigLive, LoggerLive)

const MainLive = DatabaseLive.pipe(
  Layer.provide(AppConfigLive),
  Layer.provide(ConfigLive)
)

// ---cut---
const program = Effect.gen(function* () {
  const database = yield* Database
  const result = yield* database.query("SELECT * FROM users")
  return yield* Effect.succeed(result)
})

const runnable = Effect.provide(program, MainLive)

Effect.runPromise(runnable).then(console.log)
/*
Output:
[INFO] Executing query: SELECT * FROM users
{
  result: 'Results from mysql://username:password@hostname:port/database_name'
}
*/
```

Layer Memoization

Location: 400-guides/200-context-management/400-dependency-memoization

Understand the power of layer memoization in Effect. Discover how layers can be efficiently created once and reused in the dependency graph, optimizing performance. Explore global and local memoization strategies, and learn to manually memoize layers for precise control in your Effect applications.

Layer memoization allows a layer to be created once and used multiple times in the dependency graph. If we use the same layer twice, for example

```
Layer.merge(Layer.provide(b, a), Layer.provide(c, a))
```

then the `a` layer will be allocated only once.

Memoization When Providing Globally

One important feature of an Effect application is that layers are shared by default. This means that if the same layer is used twice, and if we provide the layer globally, the layer will only be allocated a single time. For every layer in our dependency graph, there is only one instance of it that is shared between all the layers that depend on it.

For example, assume we have the three services `A`, `B`, and `C`. The implementation of both `B` and `C` is dependent on the `A` service:

```
import { Effect, Context, Layer } from "effect"

class A extends Context.Tag("A")<A, { readonly a: number }>() {}

class B extends Context.Tag("B")<B, { readonly b: string }>() {}

class C extends Context.Tag("C")<C, { readonly c: boolean }>() {}

const a = Layer.effect(
  A,
  Effect.succeed({ a: 5 }).pipe(Effect.tap(() => Effect.log("initialized")))
)

const b = Layer.effect(
  B,
  Effect.gen(function* () {
    const { a } = yield* A
    return { b: String(a) }
  })
)

const c = Layer.effect(
  C,
  Effect.gen(function* () {
    const { a } = yield* A
    return { c: a > 0 }
  })
)

const program = Effect.gen(function* () {
  yield* B
  yield* C
})

const runnable = Effect.provide(
  program,
  Layer.merge(Layer.provide(b, a), Layer.provide(c, a))
)

Effect.runPromise(runnable)
/*
Output:
timestamp=... level=INFO fiber=#2 message=initialized
*/
```

Although both `b` and `c` layers require the `a` layer, the `a` layer is instantiated only once. It is shared with both `b` and `c`.

Acquiring a Fresh Version

If we don't want to share a module, we should create a fresh, non-shared version of it through `Layer.fresh`.

```
import { Effect, Context, Layer } from "effect"

class A extends Context.Tag("A")<A, { readonly a: number }>() {}

class B extends Context.Tag("B")<B, { readonly b: string }>() {}

class C extends Context.Tag("C")<C, { readonly c: boolean }>() {}

const a = Layer.effect(
  A,
  Effect.succeed({ a: 5 }).pipe(Effect.tap(() => Effect.log("initialized")))
)

const b = Layer.effect(
  B,
  Effect.gen(function* () {
    const { a } = yield* A
    return { b: String(a) }
  })
)

const c = Layer.effect(
  C,
  Effect.gen(function* () {
    const { a } = yield* A
    return { c: a > 0 }
  })
)
```

```
const program = Effect.gen(function* () {
  yield* B
  yield* C
})

// ---cut---
const runnable = Effect.provide(
  program,
  Layer.merge(
    Layer.provide(b, Layer.fresh(a)),
    Layer.provide(c, Layer.fresh(a))
  )
)

Effect.runPromise(runnable)
/*
Output:
timestamp=... level=INFO fiber=#2 message=initialized
timestamp=... level=INFO fiber=#3 message=initialized
*/
```

No Memoization When Providing Locally

If we don't provide a layer globally but instead provide them locally, that layer doesn't support memoization by default.

In the following example, we provided the `a` layer two times locally, and Effect doesn't memoize the construction of the `a` layer. So, it will be initialized two times:

```
import { Effect, Context, Layer } from "effect"

class A extends Context.Tag("A")<A, { readonly a: number }>() {}

const a = Layer.effect(
  A,
  Effect.succeed({ a: 5 }).pipe(Effect.tap(() => Effect.log("initialized")))
)

const program = Effect.gen(function* () {
  yield* Effect.provide(A, a)
  yield* Effect.provide(A, a)
})

Effect.runPromise(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message=initialized
timestamp=... level=INFO fiber=#0 message=initialized
*/
```

Manual Memoization

We can memoize the `a` layer manually using the `Layer.memoize` operator. It will return a scoped effect that, if evaluated, will return the lazily computed result of this layer:

```
import { Effect, Context, Layer } from "effect"

class A extends Context.Tag("A")<A, { readonly a: number }>() {}

const a = Layer.effect(
  A,
  Effect.succeed({ a: 5 }).pipe(Effect.tap(() => Effect.log("initialized")))
)

const program = Effect.scoped(
  Layer.memoize(a).pipe(
    Effect.andThen((memoized) =>
      Effect.gen(function* () {
        yield* Effect.provide(A, memoized)
        yield* Effect.provide(A, memoized)
      })
    )
  )
)

Effect.runPromise(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message=initialized
*/
```

Managing Services

Location: 400-guides/200-context-management/100-services

Understand the concept of services in Effect programming, reusable components designed to provide specific capabilities across an application. Learn how to manage services with effects, create service interfaces, and provide service implementations. Explore the use of optional services for added flexibility in handling scenarios where a service may or may not be available.

In the context of programming, a **service** refers to a reusable component or functionality that can be used by different parts of an application. Services are designed to provide specific capabilities and can be shared across multiple modules or components.

Services often encapsulate common tasks or operations that are needed by different parts of an application. They can handle complex operations, interact with external systems or APIs, manage data, or perform other specialized tasks.

Services are typically designed to be modular and decoupled from the rest of the application. This allows them to be easily maintained, tested, and replaced without affecting the overall functionality of the application.

Overview

When diving into services and their integration in application development, it helps to start from the basic principles of function management and dependency handling without relying on advanced constructs. Imagine having to manually pass a service around to every function that needs it:

```
const processData = (data: Data, databaseService: DatabaseService) => {
  // Operations using the database service
}
```

This approach becomes cumbersome and unmanageable as your application grows, with services needing to be passed through multiple layers of functions.

To streamline this, you might consider using an environment object that bundles various services:

```
type Context = {
  databaseService: DatabaseService
  loggingService: LoggingService
}

const processData = (data: Data, context: Context) => {
  // Using multiple services from the context
}
```

However, this introduces a new complexity: you must ensure that the environment is correctly set up with all necessary services before it's used, which can lead to tightly coupled code and makes functional composition and testing more difficult.

The Effect library simplifies managing these dependencies by leveraging the type system. Instead of manually passing services or environment objects around, Effect allows you to declare service dependencies directly in the function's type signature using the `Requirements` parameter in the `Effect<Success, Error, Requirements>` type.

- **Dependency Declaration**: You specify what services a function needs directly in its type, pushing the complexity of dependency management into the type system.
- **Service Provision**: `Effect.provideService` is used to make a service implementation available to the functions that need it. By providing services at the start, you ensure that all parts of your application have consistent access to the required services, thus maintaining a clean and decoupled architecture.

This method abstracts the manual handling of services and dependencies, allowing developers to focus on business logic while the compiler ensures that all dependencies are correctly managed. This approach not only simplifies code but also enhances its maintainability and scalability.

Let's explore how services are managed in Effect, step by step. You'll learn the essentials:

1. **Creating a Service**: Define a service with its unique functionality and interface.
2. **Using the Service**: Access and utilize the service within your application's functions.
3. **Providing a Service Implementation**: Supply an actual implementation of the service to fulfill the declared requirements.

Managing Services with Effects

Up to this point, our examples with the Effect framework have dealt with effects that operate independently of external services. This means the `Requirements` parameter in our `Effect<Success, Error, Requirements>` type signature has been set to `never`, indicating no dependencies.

However, real-world applications often need effects that rely on specific services to function correctly. These services are managed and accessed through a construct known as `Context`.

Context serves as a repository or container for all services an effect may require. It acts like a store that maintains these services, allowing various parts of your application to access and use them as needed.

The services stored within the Context are directly reflected in the `Requirements` parameter of the `Effect` type. Each service within the Context is identified by a unique "tag," which is essentially a unique identifier for the service. When an effect needs to use a specific service, the service's tag is included in the `Requirements` type parameter.

Creating a Service

Let's start by creating a service for generating random numbers.

To create a new service, you need two things:

- A unique identifier.
- A type describing the possible operations of the service.

Let's define our first service:

- We'll use the string `"MyRandomService"` as the unique identifier.
- The service type will have a single operation called `next` that returns a random number.

```
import { Effect, Context } from "effect"

class Random extends Context.Tag("MyRandomService")<
  Random,
  { readonly next: Effect.Effect<number> }
>() {}
```

The exported `Random` value is known as a **tag** in Effect. It acts as a representation of the service and allows Effect to locate and use this service at runtime.

The service will be stored in a collection called `Context`, which can be thought of as a `Map` where the keys are tags and the values are services: `Context = Map<Tag, Service>`.

<Info> You need to specify an identifier (in this case, the string `"MyRandomService"`) to make the tag global. This ensures that two tags with the same identifier refer to the same instance.

Using a unique identifier is particularly useful in scenarios where live reloads can occur, as it helps preserve the instance across reloads. It ensures there is no duplication of instances (although it shouldn't happen, some bundlers and frameworks can behave unpredictably)

</Info>

Summary

In the Effect, understanding services, tags, and context is essential for managing requirements and building modular applications.

Concept	Description
Service	A reusable component providing specific functionality, used across different parts of an application.
Tag	A unique identifier representing a **service**, allowing Effect to locate and use it.
Context	A collection storing service, functioning like a map with **tags** as keys and **services** as values.

Using the Service

Now that we have our service tag defined, let's see how we can use it by building a simple program.

It's worth noting that the type of the `program` variable includes `Random` in the `Requirements` type parameter: `Effect<void, never, Random>`.

This indicates that our program requires the `Random` service to be provided in order to execute successfully.

If we attempt to execute the effect without providing the necessary service we will encounter a type-checking error:

```
// @errors: 2345
import { Effect, Context } from "effect"

class Random extends Context.Tag("MyRandomService")<
  Random,
  { readonly next: Effect.Effect<number> }
>() {}

const program = Effect.gen(function* () {
  const random = yield* Random
  const randomNumber = yield* random.next
  console.log(`random number: ${randomNumber}`)
})

// ---cut---
Effect.runSync(program)
```

To resolve this error and successfully execute the program, we need to provide an actual implementation of the `Random` service.

In the next section, we will explore how to implement and provide the `Random` service to our program, enabling us to run it successfully.

Providing a Service Implementation

In order to provide an actual implementation of the `Random` service, we can utilize the `Effect.provideService` function.

```
import { Effect, Context } from "effect"

class Random extends Context.Tag("MyRandomService")<
  Random,
  { readonly next: Effect.Effect<number> }
>() {}

const program = Effect.gen(function* () {
```

```
  const random = yield* Random
  const randomNumber = yield* random.next
  console.log(`random number: ${randomNumber}`)
})

// ---cut---
const runnable = Effect.provideService(program, Random, {
  next: Effect.sync(() => Math.random())
})

Effect.runPromise(runnable)
/*
Output:
random number: 0.8241872233134417
*/
```

In the code snippet above, we call the `program` that we defined earlier and provide it with an implementation of the `Random` service. We use the `Effect.provideService` function to associate the `Random` tag with its implementation, an object with a `next` operation that generates a random number.

Notice that the `Requirements` type parameter of the `runnable` effect is now `never`. This indicates that the effect no longer requires any service to be provided. With the implementation of the `Random` service in place, we are able to run the program without any further requirements.

Extracting the Service Type

To retrieve the service type from a tag, use the `Context.Tag.Service` utility type:

```
import { Effect, Context } from "effect"

class Random extends Context.Tag("MyRandomService")<
  Random,
  { readonly next: Effect.Effect<number> }
>() {}

type RandomShape = Context.Tag.Service<Random>
/*
This is equivalent to:
type RandomShape = {
    readonly next: Effect.Effect<number>;
}
*/
```

Using Multiple Services

When we require the usage of more than one service, the process remains similar to what we've learned in defining a service, repeated for each service needed. Let's examine an example where we need two services, namely `Random` and `Logger`:

The `program` effect now has a `Requirements` type parameter of `Random | Logger`, indicating that it requires both the `Random` and `Logger` services to be provided.

To execute the `program`, we need to provide implementations for both services:

```
import { Effect, Context } from "effect"

class Random extends Context.Tag("MyRandomService")<
  Random,
  {
    readonly next: Effect.Effect<number>
  }
>() {}

class Logger extends Context.Tag("MyLoggerService")<
  Logger,
  {
    readonly log: (message: string) => Effect.Effect<void>
  }
>() {}

const program = Effect.gen(function* () {
  const random = yield* Random
  const logger = yield* Logger
  const randomNumber = yield* random.next
  return yield* logger.log(String(randomNumber))
})

// ---cut---
// Provide service implementations for 'Random' and 'Logger'
const runnable1 = program.pipe(
  Effect.provideService(Random, {
    next: Effect.sync(() => Math.random())
  }),
  Effect.provideService(Logger, {
    log: (message) => Effect.sync(() => console.log(message))
  })
)
```

Alternatively, instead of calling `provideService` multiple times, we can combine the service implementations into a single `Context` and then provide the entire context using the `Effect.provide` function:

```
import { Effect, Context } from "effect"

class Random extends Context.Tag("MyRandomService")<
  Random,
  {
    readonly next: Effect.Effect<number>
  }
>() {}

class Logger extends Context.Tag("MyLoggerService")<
  Logger,
  {
    readonly log: (message: string) => Effect.Effect<void>
  }
>() {}

const program = Effect.gen(function* () {
  const random = yield* Random
  const logger = yield* Logger
  const randomNumber = yield* random.next
  return yield* logger.log(String(randomNumber))
})

// ---cut---
// Combine service implementations into a single 'Context'
const context = Context.empty().pipe(
  Context.add(Random, { next: Effect.sync(() => Math.random()) }),
  Context.add(Logger, {
    log: (message) => Effect.sync(() => console.log(message))
  })
)

// Provide the entire context to the 'program'
const runnable2 = Effect.provide(program, context)
```

By providing the necessary implementations for each service, we ensure that the runnable effect can access and utilize both services when it is executed.

Optional Services

There are situations where we may want to access a service implementation only if it is available. In such cases, we can use the `Effect.serviceOption` function to handle this scenario.

The `Effect.serviceOption` function returns an implementation that is available only if it is actually provided before executing this effect. To represent this optionality it returns an Option of the implementation.

Let's take a look at an example that demonstrates the usage of optional services:

In the code above, we can observe that the `Requirements` type parameter of the `program` effect is `never`, even though we are working with a service. This allows us to access something from the context only if it is actually provided before executing this effect.

When we run the `program` effect without providing the `Random` service:

```
Effect.runPromise(program).then(console.log)
// Output: -1
```

We see that the log message contains `-1`, which is the default value we provided when the service was not available.

However, if we provide the `Random` service implementation:

```
Effect.runPromise(
  Effect.provideService(program, Random, {
    next: Effect.sync(() => Math.random())
  })
).then(console.log)
// Output: 0.9957979486841035
```

We can observe that the log message now contains a random number generated by the `next` operation of the `Random` service.

Streaming

Location: 400-guides/660-streaming/index

Streaming

Sinks

Location: 400-guides/660-streaming/200-sink/index

Sinks

Leftovers

Location: 400-guides/660-streaming/200-sink/500-leftovers

Explore handling unconsumed elements with sinks. Learn to collect or ignore leftovers using `Sink.collectLeftover` and `Sink.ignoreLeftover`. Efficiently manage and process remaining elements from upstream sources in data streams.

In this section, we'll explore how to deal with elements that may be left unconsumed by sinks. Sinks can consume varying numbers of elements from their upstream, and we'll learn how to collect or ignore any leftovers.

Collecting Leftovers

When a sink consumes elements from an upstream source, it may not use all of them. These unconsumed elements are referred to as "leftovers." To collect these leftovers, we can use `Sink.collectLeftover`. It returns a tuple containing the result of the previous sink operation and any leftover elements:

```
import { Stream, Sink, Effect } from "effect"

const s1 = Stream.make(1, 2, 3, 4, 5).pipe(
  Stream.run(Sink.take<number>(3).pipe(Sink.collectLeftover))
)

Effect.runPromise(s1).then(console.log)
/*
Output:
[
  {
    _id: "Chunk",
    values: [ 1, 2, 3 ]
  }, {
    _id: "Chunk",
    values: [ 4, 5 ]
  }
]
*/

const s2 = Stream.make(1, 2, 3, 4, 5).pipe(
  Stream.run(Sink.head<number>().pipe(Sink.collectLeftover))
)

Effect.runPromise(s2).then(console.log)
/*
Output:
[
  {
    _id: "Option",
    _tag: "Some",
    value: 1
  }, {
    _id: "Chunk",
    values: [ 2, 3, 4, 5 ]
  }
]
*/
```

Ignoring Leftovers

When leftover elements are not needed, they can be ignored using `Sink.ignoreLeftover`:

```
import { Stream, Sink, Effect } from "effect"

const s1 = Stream.make(1, 2, 3, 4, 5).pipe(
  Stream.run(
    Sink.take<number>(3).pipe(Sink.ignoreLeftover).pipe(Sink.collectLeftover)
  )
)

Effect.runPromise(s1).then(console.log)
/*
Output:
[
  {
    _id: "Chunk",
    values: [ 1, 2, 3 ]
  }, {
    _id: "Chunk",
    values: []
  }
```

```
]
*/
```

Creating Sinks

Location: 400-guides/660-streaming/200-sink/200-creating

Learn how to construct powerful sinks for processing stream elements. Explore common constructors like `head`, `last`, `count`, `sum`, `take`, `drain`, `timed`, `forEach`, and discover how to create sinks from success and failure. Dive into collecting strategies using `collectAll`, `collectAllToSet`, `collectAllToMap`, `collectAllN`, `collectAllWhile`, `collectAllToSetN`, `collectAllToMapN`, folding techniques with `foldLeft`, `fold`, `foldWeighted`, `foldUntil`, and more.

In the world of streams, sinks are used to consume and process the elements of a stream. Here, we will introduce some common sink constructors that allow you to create sinks for specific tasks.

Common Constructors

head

The `head` sink creates a sink that captures the first element of a stream. If the stream is empty, it returns `None`.

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.head()))

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "Option",
  _tag: "Some",
  value: 1
}
*/
```

last

The `last` sink consumes all elements of a stream and returns the last element of the stream.

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.last()))

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "Option",
  _tag: "Some",
  value: 4
}
*/
```

count

The `count` sink consumes all elements of the stream and counts the number of elements fed to it.

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.count))

Effect.runPromise(effect).then(console.log)
/*
Output:
4
*/
```

sum

The `sum` sink consumes all elements of the stream and sums incoming numeric values.

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.sum))

Effect.runPromise(effect).then(console.log)
/*
Output:
10
*/
```

take

The `take` sink takes the specified number of values from the stream and results in a Chunk data type.

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.take(3)))

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3 ]
}
*/
```

drain

The `drain` sink ignores its inputs, effectively discarding them.

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.drain))

Effect.runPromise(effect).then(console.log)
/*
Output:
undefined
*/
```

timed

The `timed` sink executes the stream and measures its execution time, providing the duration.

```
import { Stream, Schedule, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(
  Stream.schedule(Schedule.spaced("100 millis")),
  Stream.run(Sink.timed)
)

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "Duration",
  _tag: "Millis",
  millis: 523
}
*/
```

forEach

The `forEach` sink executes the provided effectful function for every element fed to it.

```
import { Stream, Sink, Console, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(
  Stream.run(Sink.forEach(Console.log))
)

Effect.runPromise(effect).then(console.log)
/*
Output:
1
2
3
4
undefined
*/
```

From Success and Failure

In the realm of data streams, similar to crafting streams to hold and manipulate data, we can also create sinks using the `Sink.fail` and `Sink.succeed` functions.

Succeeding Sink

Let's start with a sink that doesn't consume any elements from its upstream but instead succeeds with a numeric value:

```
import { Stream, Sink, Effect } from "effect"
```

```
const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.succeed(0)))

Effect.runPromise(effect).then(console.log)
/*
Output:
0
*/
```

Failing Sink

Now, consider a sink that also doesn't consume any elements from its upstream but deliberately fails, generating an error message of type `string`:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.fail("fail!")))

Effect.runPromiseExit(effect).then(console.log)
/*
Output:
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: 'fail!' }
}
*/
```

Collecting

Collecting All Elements

To gather all the elements from a data stream into a Chunk, you can employ the `Sink.collectAll()` function:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(Stream.run(Sink.collectAll()))

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, 4 ]
}
*/
```

Collecting into a HashSet

If you want to accumulate the elements into a `HashSet`, you can use `Sink.collectAllToSet()`. This function ensures that each element appears only once in the resulting set:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 2, 3, 4, 4).pipe(
  Stream.run(Sink.collectAllToSet())
)

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "HashSet",
  values: [ 1, 2, 3, 4 ]
}
*/
```

Collecting into a HashMap

For more advanced collection needs, you can use `Sink.collectAllToMap()`. This function allows you to accumulate and merge elements into a `HashMap<K, A>` using a specified merge function. In the following example, we determine map keys with `(n) => n % 3` and merge elements with the same key using `(a, b) => a + b`:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 3, 2, 3, 1, 5, 1).pipe(
  Stream.run(
    Sink.collectAllToMap(
      (n) => n % 3,
      (a, b) => a + b
    )
  )
)

Effect.runPromise(effect).then(console.log)
/*
```

```
Output:
{
  _id: "HashMap",
  values: [
    [ 0, 6 ], [ 1, 3 ], [ 2, 7 ]
  ]
}
*/
```

Collecting a Specified Number

If you only want to collect a specific number of elements from a stream into a Chunk, you can use `Sink.collectAllN(n)`:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4, 5).pipe(
  Stream.run(Sink.collectAllN(3))
)

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3 ]
}
*/
```

Collecting While Meeting a Condition

To accumulate elements as long as they satisfy a specific condition, you can use `Sink.collectAllWhile(predicate)`. This function will keep gathering elements until the predicate returns `false`:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 0, 4, 0, 6, 7).pipe(
  Stream.run(Sink.collectAllWhile((n) => n !== 0))
)

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2 ]
}
*/
```

Collecting into HashSets of a Specific Size

For more controlled collection into sets with a maximum size of `n`, you can utilize `Sink.collectAllToSetN(n)`:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 2, 3, 4, 4).pipe(
  Stream.run(Sink.collectAllToSetN(3))
)

Effect.runPromise(effect).then(console.log)
/*
Output:
{
  _id: "HashSet",
  values: [ 1, 2, 3 ]
}
*/
```

Collecting into HashMaps with Limited Keys

If you need to accumulate elements into maps with a maximum of `n` keys, you can employ `Sink.collectAllToMapN(n, keyFunction, mergeFunction)`:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 3, 2, 3, 1, 5, 1).pipe(
  Stream.run(
    Sink.collectAllToMapN(
      3,
      (n) => n,
      (a, b) => a + b
    )
  )
)

Effect.runPromise(effect).then(console.log)
```

```
/*
Output:
{
  _id: "HashMap",
  values: [
    [ 1, 2 ], [ 2, 2 ], [ 3, 6 ]
  ]
}
*/
```

Folding

Folding Left

Imagine you have a stream of numbers, and you want to reduce them into a single value by applying an operation to each element sequentially. You can achieve this using the `Sink.foldLeft` function:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4).pipe(
  Stream.run(Sink.foldLeft(0, (a, b) => a + b))
)

Effect.runPromise(effect).then(console.log)
/*
Output:
10
*/
```

Folding with Termination

In some cases, you may want to fold elements in a stream but stop the folding process when a certain condition is met. This is called "short-circuiting." You can achieve this using the `Sink.fold` function, which allows you to specify a termination predicate:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.iterate(0, (n) => n + 1).pipe(
  Stream.run(
    Sink.fold(
      0,
      (sum) => sum <= 10,
      (a, b) => a + b
    )
  )
)

Effect.runPromise(effect).then(console.log)
/*
Output:
15
*/
```

Folding with Weighted Elements

Sometimes, you may want to fold elements based on their weight or cost, accumulating them until a certain maximum cost is reached. You can do this using `Sink.foldWeighted`. In the following example, we group elements based on a weight of 1, restarting the folding process when the total weight reaches 3:

```
import { Stream, Sink, Chunk, Effect } from "effect"

const stream = Stream.make(3, 2, 4, 1, 5, 6, 2, 1, 3, 5, 6).pipe(
  Stream.transduce(
    Sink.foldWeighted({
      initial: Chunk.empty<number>(),
      maxCost: 3,
      cost: () => 1,
      body: (acc, el) => Chunk.append(acc, el)
    })
  )
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    {
      _id: "Chunk",
      values: [ 3, 2, 4 ]
    }, {
      _id: "Chunk",
      values: [ 1, 5, 6 ]
    }, {
      _id: "Chunk",
```

```
      values: [ 2, 1, 3 ]
    }, {
      _id: "Chunk",
      values: [ 5, 6 ]
    }
  ]
}
*/
```

Folding Until a Limit

If you want to fold elements up to a specific limit, you can use `Sink.foldUntil`. In the following example, we fold elements until we have accumulated 3 of them:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).pipe(
  Stream.run(Sink.foldUntil(0, 3, (a, b) => a + b))
)

Effect.runPromise(effect).then(console.log)
/*
Output:
6
*/
```

Sink Operations

Location: 400-guides/660-streaming/200-sink/300-operations

Explore sink operations to transform or filter their behavior. Learn to adapt sinks for different input types using `Sink.mapInput`. Discover how `Sink.dimap` allows complete conversion between input and output types. Utilize `Sink.filterInput` to selectively process elements based on specific conditions.

In the previous sections, we learned how to create and use sinks. Now, let's explore some operations you can perform on sinks to transform or filter their behavior.

Changing the Input

Sometimes, you have a sink that works perfectly with one type of input, but you want to use it with a different type. This is where `Sink.mapInput` comes in handy. While `Sink.map` modifies the output of a function, `Sink.mapInput` modifies the input. It allows you to adapt your sink to work with a different input.

Imagine you have a `Sink.sum` that calculates the sum of incoming numeric values. However, your stream contains strings, not numbers. You can use `mapInput` to convert your strings into numbers and make `Sink.sum` compatible with your stream:

```
import { Stream, Sink, Effect } from "effect"

const numericSum = Sink.sum

const stringSum = numericSum.pipe(
  Sink.mapInput((s: string) => Number.parseFloat(s))
)

Effect.runPromise(
  Stream.make("1", "2", "3", "4", "5").pipe(Stream.run(stringSum))
).then(console.log)
/*
Output:
15
*/
```

Transforming Both Input and Output

If you need to change both the input and output of a sink, you can use `Sink.dimap`. It's an extended version of `mapInput` that lets you transform both types. This can be useful when you need to perform a complete conversion between your input and output types:

```
import { Stream, Sink, Effect } from "effect"

// Convert its input to integers, do the computation and then convert them back to a string
const sumSink = Sink.sum.pipe(
  Sink.dimap({
    onInput: (s: string) => Number.parseFloat(s),
    onDone: (n) => String(n)
  })
)

Effect.runPromise(
  Stream.make("1", "2", "3", "4", "5").pipe(Stream.run(sumSink))
).then(console.log)
/*
```

```
Output:
15 <-- as string
*/
```

Filtering Input

Sinks offer a way to filter incoming elements using `Sink.filterInput`. This allows you to collect or process only the elements that meet a specific condition. In the following example, we collect elements in chunks of three and filter out the negative numbers:

```
import { Stream, Sink, Effect } from "effect"

const stream = Stream.make(1, -2, 0, 1, 3, -3, 4, 2, 0, 1, -3, 1, 1, 6).pipe(
  Stream.transduce(
    Sink.collectAllN<number>(3).pipe(Sink.filterInput((n) => n > 0))
  )
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    {
      _id: "Chunk",
      values: [ 1, 1, 3 ]
    }, {
      _id: "Chunk",
      values: [ 4, 2, 1 ]
    }, {
      _id: "Chunk",
      values: [ 1, 1, 6 ]
    }, {
      _id: "Chunk",
      values: []
    }
  ]
}
*/
```

Introduction to Sinks

Location: 400-guides/660-streaming/200-sink/100-introduction

Explore the role of `Sink` in stream processing. Learn how a `Sink` consumes elements, handles errors, produces values, and manages leftover elements. Use it seamlessly with `Stream.run` for efficient stream processing.

In the world of streams, a `Sink<A, In, L, E, R>` plays a crucial role. It's like a specialized function designed to consume elements generated by a `Stream`. Here's a breakdown of what a `Sink` does:

- It can consume a varying number of `In` elements, which might be zero, one, or more.
- It has the potential to encounter errors of type `E`.
- Ultimately, it produces a value of type `A`.
- Additionally, it can return a remainder of type `L`, which represents any leftover elements.

To use a `Sink` for processing a stream, you simply pass it to the `Stream.run` function:

```
import { Stream, Sink, Effect } from "effect"

const stream = Stream.make(1, 2, 3) // Define a stream with numbers 1, 2, and 3

const sink = Sink.sum // Choose a sink that sums up numbers

const sum = Stream.run(stream, sink) // Run the stream through the sink

Effect.runPromise(sum).then(console.log)
/*
Output:
6
*/
```

Parallel Operators

Location: 400-guides/660-streaming/200-sink/400-parallel-operators

Explore parallel operations like `Sink.zip` for combining results and `Sink.race` for racing concurrent sinks. Learn how to run multiple sinks concurrently, combining or selecting the first to complete. Enhance task performance by executing operations simultaneously.

In this section, we'll explore parallel operations that allow you to run multiple sinks concurrently. These operations can be quite useful when you need to perform tasks simultaneously.

Parallel Zipping: Combining Results

When you want to run two sinks concurrently and combine their results, you can use `Sink.zip`. This operation runs both sinks concurrently and combines their outcomes into a tuple:

```
import { Sink, Console, Stream, Schedule, Effect } from "effect"

const s1 = Sink.forEach((s: string) => Console.log(`sink 1: ${s}`)).pipe(
  Sink.as(1)
)

const s2 = Sink.forEach((s: string) => Console.log(`sink 2: ${s}`)).pipe(
  Sink.as(2)
)

const sink = s1.pipe(Sink.zip(s2, { concurrent: true }))

Effect.runPromise(
  Stream.make("1", "2", "3", "4", "5").pipe(
    Stream.schedule(Schedule.spaced("10 millis")),
    Stream.run(sink)
  )
).then(console.log)
/*
Output:
sink 1: 1
sink 2: 1
sink 1: 2
sink 2: 2
sink 1: 3
sink 2: 3
sink 1: 4
sink 2: 4
sink 1: 5
sink 2: 5
[ 1, 2 ]
*/
```

Racing: First One Wins

Another useful operation is `Sink.race`, which lets you race multiple sinks concurrently. The sink that completes first will provide the result for your program:

```
import { Sink, Console, Stream, Schedule, Effect } from "effect"

const s1 = Sink.forEach((s: string) => Console.log(`sink 1: ${s}`)).pipe(
  Sink.as(1)
)

const s2 = Sink.forEach((s: string) => Console.log(`sink 2: ${s}`)).pipe(
  Sink.as(2)
)

const sink = s1.pipe(Sink.race(s2))

Effect.runPromise(
  Stream.make("1", "2", "3", "4", "5").pipe(
    Stream.schedule(Schedule.spaced("10 millis")),
    Stream.run(sink)
  )
).then(console.log)
/*
Output:
sink 1: 1
sink 2: 1
sink 1: 2
sink 2: 2
sink 1: 3
sink 2: 3
sink 1: 4
sink 2: 4
sink 1: 5
sink 2: 5
1
*/
```

Consuming Streams

Location: 400-guides/660-streaming/100-stream/500-consuming-streams

Consume streams effectively using methods like `runCollect` to gather elements into a single `Chunk`, `runForEach` to process elements with a callback, `fold` for performing operations, and `Sink` for specialized consumption. Learn key techniques for working with streams in your applications.

When working with streams, it's essential to understand how to consume the data they produce. In this guide, we'll walk through several common methods for consuming streams.

Using `runCollect`

To gather all the elements from a stream into a single `Chunk`, you can use the `Stream.runCollect` function.

```
import { Stream, Effect } from "effect"

const stream = Stream.make(1, 2, 3, 4, 5)

const collectedData = Stream.runCollect(stream)

Effect.runPromise(collectedData).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, 4, 5 ]
}
*/
```

Using `runForEach`

Another way to consume elements of a stream is by using `Stream.runForEach`. It takes a callback function that receives each element of the stream. Here's an example:

```
import { Stream, Effect, Console } from "effect"

const effect = Stream.make(1, 2, 3).pipe(
  Stream.runForEach((n) => Console.log(n))
)

Effect.runPromise(effect).then(console.log)
/*
Output:
1
2
3
undefined
*/
```

In this example, we use `Stream.runForEach` to log each element to the console.

Using a Fold Operation

The `Stream.fold` function is another way to consume a stream by performing a fold operation over the stream of values and returning an effect containing the result. Here are a couple of examples:

```
import { Stream, Effect } from "effect"

const e1 = Stream.make(1, 2, 3, 4, 5).pipe(Stream.runFold(0, (a, b) => a + b))

Effect.runPromise(e1).then(console.log) // Output: 15

const e2 = Stream.make(1, 2, 3, 4, 5).pipe(
  Stream.runFoldWhile(
    0,
    (n) => n <= 3,
    (a, b) => a + b
  )
)

Effect.runPromise(e2).then(console.log) // Output: 6
```

In the first example (`e1`), we use `Stream.runFold` to calculate the sum of all elements. In the second example (`e2`), we use `Stream.runFoldWhile` to calculate the sum but only until a certain condition is met.

Using a Sink

To consume a stream using a Sink, you can pass the `Sink` to the `Stream.run` function. Here's an example:

```
import { Stream, Sink, Effect } from "effect"

const effect = Stream.make(1, 2, 3).pipe(Stream.run(Sink.sum))

Effect.runPromise(effect).then(console.log) // Output: 6
```

In this example, we use a `Sink` to calculate the sum of the elements in the stream.

Error Handling in Streams

Location: 400-guides/660-streaming/100-stream/600-error-handling

Effectively handle errors in streams using functions like `orElse` for seamless recovery, `catchAll` for advanced error handling, and `retry` to handle temporary failures. Learn to refine errors, set timeouts with various operators, and gracefully recover from defects, ensuring robust stream processing in your applications.

Recovering from Failure

When working with streams that may encounter errors, it's crucial to know how to handle these errors gracefully. The `Stream.orElse` function is a powerful tool for recovering from failures and switching to an alternative stream in case of an error.

Here's a practical example:

```
import { Stream, Effect } from "effect"

const s1 = Stream.make(1, 2, 3).pipe(
  Stream.concat(Stream.fail("Oh! Error!")),
  Stream.concat(Stream.make(4, 5))
)

const s2 = Stream.make("a", "b", "c")

const stream = Stream.orElse(s1, () => s2)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, "a", "b", "c" ]
}
*/
```

In this example, `s1` encounters an error, but instead of terminating the stream, we gracefully switch to `s2` using `Stream.orElse`. This ensures that we can continue processing data even if one stream fails.

There's also a variant called `Stream.orElseEither` that uses the Either data type to distinguish elements from the two streams based on success or failure:

```
import { Stream, Effect } from "effect"

const s1 = Stream.make(1, 2, 3).pipe(
  Stream.concat(Stream.fail("Oh! Error!")),
  Stream.concat(Stream.make(4, 5))
)

const s2 = Stream.make("a", "b", "c")

const stream = Stream.orElseEither(s1, () => s2)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    {
      _id: "Either",
      _tag: "Left",
      left: 1
    }, {
      _id: "Either",
      _tag: "Left",
      left: 2
    }, {
      _id: "Either",
      _tag: "Left",
      left: 3
    }, {
      _id: "Either",
      _tag: "Right",
      right: "a"
    }, {
      _id: "Either",
      _tag: "Right",
      right: "b"
    }, {
      _id: "Either",
      _tag: "Right",
      right: "c"
    }
  ]
```

```
}
*/
```

The `Stream.catchAll` function provides advanced error handling capabilities compared to `Stream.orElse`. With `Stream.catchAll`, you can make decisions based on both the type and value of the encountered failure.

```
import { Stream, Effect } from "effect"

const s1 = Stream.make(1, 2, 3).pipe(
  Stream.concat(Stream.fail("Uh Oh!" as const)),
  Stream.concat(Stream.make(4, 5)),
  Stream.concat(Stream.fail("Ouch" as const))
)

const s2 = Stream.make("a", "b", "c")

const s3 = Stream.make(true, false, false)

const stream = Stream.catchAll(
  s1,
  (error): Stream.Stream<string | boolean> => {
    switch (error) {
      case "Uh Oh!":
        return s2
      case "Ouch":
        return s3
    }
  }
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, "a", "b", "c" ]
}
*/
```

In this example, we have a stream, `s1`, which may encounter two different types of errors. Instead of a straightforward switch to an alternative stream, as done with `Stream.orElse`, we employ `Stream.catchAll` to precisely determine how to handle each type of error. This level of control over error recovery enables you to choose different streams or actions based on the specific error conditions.

Recovering from Defects

When working with streams, it's essential to be prepared for various failure scenarios, including defects that might occur during stream processing. To address this, the `Stream.catchAllCause` function provides a robust solution. It enables you to gracefully handle and recover from any type of failure that may arise.

Here's an example to illustrate its usage:

```
import { Stream, Effect } from "effect"

const s1 = Stream.make(1, 2, 3).pipe(
  Stream.concat(Stream.dieMessage("Boom!")),
  Stream.concat(Stream.make(4, 5))
)

const s2 = Stream.make("a", "b", "c")

const stream = Stream.catchAllCause(s1, () => s2)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, "a", "b", "c" ]
}
*/
```

In this example, `s1` may encounter a defect, but instead of crashing the application, we use `Stream.catchAllCause` to gracefully switch to an alternative stream, `s2`. This ensures that your application remains robust and continues processing data even in the face of unexpected issues.

Recovery from Some Errors

In stream processing, there may be situations where you need to recover from specific types of failures. The `Stream.catchSome` and `Stream.catchSomeCause` functions come to the rescue, allowing you to handle and mitigate errors selectively.

If you want to recover from a particular error, you can use `Stream.catchSome`:

```
import { Stream, Effect, Option } from "effect"

const s1 = Stream.make(1, 2, 3).pipe(
  Stream.concat(Stream.fail("Oh! Error!")),
```

```
  Stream.concat(Stream.make(4, 5))
)

const s2 = Stream.make("a", "b", "c")

const stream = Stream.catchSome(s1, (error) => {
  if (error === "Oh! Error!") {
    return Option.some(s2)
  }
  return Option.none()
})

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, "a", "b", "c" ]
}
*/
```

To recover from a specific cause, you can use the `Stream.catchSomeCause` function:

```
import { Stream, Effect, Option, Cause } from "effect"

const s1 = Stream.make(1, 2, 3).pipe(
  Stream.concat(Stream.dieMessage("Oh! Error!")),
  Stream.concat(Stream.make(4, 5))
)

const s2 = Stream.make("a", "b", "c")

const stream = Stream.catchSomeCause(s1, (cause) => {
  if (Cause.isDie(cause)) {
    return Option.some(s2)
  }
  return Option.none()
})

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, "a", "b", "c" ]
}
*/
```

Recovering to Effect

In stream processing, it's crucial to handle errors gracefully and perform cleanup tasks when needed. The `Stream.onError` function allows us to do just that. If our stream encounters an error, we can specify a cleanup task to be executed.

```
import { Stream, Console, Effect } from "effect"

const stream = Stream.make(1, 2, 3).pipe(
  Stream.concat(Stream.dieMessage("Oh! Boom!")),
  Stream.concat(Stream.make(4, 5)),
  Stream.onError(() =>
    Console.log(
      "Stream application closed! We are doing some cleanup jobs."
    ).pipe(Effect.orDie)
  )
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
Stream application closed! We are doing some cleanup jobs.
error: RuntimeException: Oh! Boom!
*/
```

Retry a Failing Stream

Sometimes, streams may encounter failures that are temporary or recoverable. In such cases, the `Stream.retry` operator comes in handy. It allows you to specify a retry schedule, and the stream will be retried according to that schedule.

Here's an example to illustrate how it works:

```
// @types: node
import { Stream, Effect, Schedule } from "effect"
import * as NodeReadLine from "node:readline"

const stream = Stream.make(1, 2, 3).pipe(
  Stream.concat(
    Stream.fromEffect(
```

```
      Effect.gen(function* () {
        const s = yield* readLine("Enter a number: ")
        const n = parseInt(s)
        if (Number.isNaN(n)) {
          return yield* Effect.fail("NaN")
        }
        return n
      })
    ).pipe(Stream.retry(Schedule.exponential("1 second")))
  )
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
Enter a number: a
Enter a number: b
Enter a number: c
Enter a number: 4
{
  _id: "Chunk",
  values: [ 1, 2, 3, 4 ]
}
*/

const readLine = (message: string): Effect.Effect<string> =>
  Effect.promise(
    () =>
      new Promise((resolve) => {
        const rl = NodeReadLine.createInterface({
          input: process.stdin,
          output: process.stdout
        })
        rl.question(message, (answer) => {
          rl.close()
          resolve(answer)
        })
      })
  )
```

In this example, the stream asks the user to input a number, but if an invalid value is entered (e.g., "a," "b," "c"), it fails with "NaN." However, we use `Stream.retry` with an exponential backoff schedule, which means it will retry after a delay of increasing duration. This allows us to handle temporary errors and eventually collect valid input.

Refining Errors

When working with streams, there might be situations where you want to selectively keep certain errors and terminate the stream with the remaining errors. You can achieve this using the `Stream.refineOrDie` function.

Here's an example to illustrate how it works:

```
import { Stream, Option } from "effect"

const stream = Stream.fail(new Error())

const res = Stream.refineOrDie(stream, (error) => {
  if (error instanceof SyntaxError) {
    return Option.some(error)
  }
  return Option.none()
})
```

In this example, `stream` initially fails with a generic `Error`. However, we use `Stream.refineOrDie` to filter and keep only errors of type `SyntaxError`. Any other errors will be terminated, while `SyntaxErrors` will be retained in `refinedStream`.

Timing Out

When working with streams, there are scenarios where you may want to handle timeouts, such as terminating a stream if it doesn't produce a value within a certain duration. In this section, we'll explore how to manage timeouts using various operators.

timeout

The `Stream.timeout` operator allows you to set a timeout on a stream. If the stream does not produce a value within the specified duration, it terminates.

```
import { Stream, Effect } from "effect"

const stream = Stream.fromEffect(Effect.never).pipe(
  Stream.timeout("2 seconds")
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
{
```

```
  _id: "Chunk",
  values: []
}
*/
```

timeoutFail

The `Stream.timeoutFail` operator combines a timeout with a custom failure message. If the stream times out, it fails with the specified error message.

```
import { Stream, Effect } from "effect"

const stream = Stream.fromEffect(Effect.never).pipe(
  Stream.timeoutFail(() => "timeout", "2 seconds")
)

Effect.runPromiseExit(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: 'timeout' }
}
*/
```

timeoutFailCause

Similar to `Stream.timeoutFail`, `Stream.timeoutFailCause` combines a timeout with a custom failure cause. If the stream times out, it fails with the specified cause.

```
import { Stream, Effect, Cause } from "effect"

const stream = Stream.fromEffect(Effect.never).pipe(
  Stream.timeoutFailCause(() => Cause.die("timeout"), "2 seconds")
)

Effect.runPromiseExit(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Die', defect: 'timeout' }
}
*/
```

timeoutTo

The `Stream.timeoutTo` operator allows you to switch to another stream if the first stream does not produce a value within the specified duration.

```
import { Stream, Effect } from "effect"

const stream = Stream.fromEffect(Effect.never).pipe(
  Stream.timeoutTo("2 seconds", Stream.make(1, 2, 3))
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
{
  _id: "Chunk",
  values: [ 1, 2, 3 ]
}
*/
```

Streams

Location: 400-guides/660-streaming/100-stream/index

Streams

Creating Streams

Location: 400-guides/660-streaming/100-stream/200-creating

Explore diverse methods for crafting `Stream`s in Effect, tailored to your specific needs. Learn about common constructors like `make`, `empty`, `unit`, `range`, `iterate`, and `scoped`. Discover how to generate streams from success and failure using `succeed` and `fail` functions, and construct streams from chunks, effects, asynchronous callbacks, iterables, repetitions, unfolding, pagination, queues, pub/sub, and schedules. Dive into practical examples and gain insights into the nuances of each method, enabling you to harness the full power of Effect's streaming capabilities.

In this section, we'll explore various methods for creating Effect `Stream`s. These methods will help you generate streams tailored to your needs.

Common Constructors

make

You can create a pure stream by using the `Stream.make` constructor. This constructor accepts a variable list of values as its arguments.

```
import { Stream } from "effect"

const stream = Stream.make(1, 2, 3)
```

empty

Sometimes, you may require a stream that doesn't produce any values. In such cases, you can use `Stream.empty`. This constructor creates a stream that remains empty.

```
import { Stream } from "effect"

const stream = Stream.empty
```

void

If you need a stream that contains a single `void` value, you can use `Stream.void`. This constructor is handy when you want to represent a stream with a single event or signal.

```
import { Stream } from "effect"

const stream = Stream.void
```

range

To create a stream of integers within a specified range `[min, max]` (including both endpoints, `min` and `max`), you can use `Stream.range`. This is particularly useful for generating a stream of sequential numbers.

```
import { Stream } from "effect"

// Creating a stream of numbers from 1 to 5
const stream = Stream.range(1, 5) // Produces 1, 2, 3, 4, 5
```

iterate

With `Stream.iterate`, you can generate a stream by applying a function iteratively to an initial value. The initial value becomes the first element produced by the stream, followed by subsequent values produced by `f(init)`, `f(f(init))`, and so on.

```
import { Stream } from "effect"

// Creating a stream of incrementing numbers
const stream = Stream.iterate(1, (n) => n + 1) // Produces 1, 2, 3, ...
```

scoped

`Stream.scoped` is used to create a single-valued stream from a scoped resource. It can be handy when dealing with resources that require explicit acquisition, usage, and release.

```
import { Stream, Effect, Console } from "effect"

// Creating a single-valued stream from a scoped resource
const stream = Stream.scoped(
  Effect.acquireUseRelease(
    Console.log("acquire"),
    () => Console.log("use"),
    () => Console.log("release")
  )
)
```

From Success and Failure

Much like the `Effect` data type, you can generate a `Stream` using the `fail` and `succeed` functions:

```
import { Stream } from "effect"

// Creating a stream that can emit errors
const streamWithError: Stream.Stream<never, string> = Stream.fail("Uh oh!")

// Creating a stream that emits a numeric value
const streamWithNumber: Stream.Stream<number> = Stream.succeed(5)
```

From Chunks

You can construct a stream from a `Chunk` like this:

```
import { Stream, Chunk } from "effect"

// Creating a stream with values from a single Chunk
const stream = Stream.fromChunk(Chunk.make(1, 2, 3))
```

Moreover, you can create a stream from multiple `Chunk`s as well:

```
import { Stream, Chunk } from "effect"

// Creating a stream with values from multiple Chunks
const stream = Stream.fromChunks(Chunk.make(1, 2, 3), Chunk.make(4, 5, 6))
```

From Effect

You can generate a stream from an Effect workflow by employing the `Stream.fromEffect` constructor. For instance, consider the following stream, which generates a single random number:

```
import { Stream, Random } from "effect"

const stream = Stream.fromEffect(Random.nextInt)
```

This method allows you to seamlessly transform the output of an Effect into a stream, providing a straightforward way to work with asynchronous operations within your streams.

From Asynchronous Callback

Imagine you have an asynchronous function that relies on callbacks. If you want to capture the results emitted by those callbacks as a stream, you can use the `Stream.async` function. This function is designed to adapt functions that invoke their callbacks multiple times and emit the results as a stream.

Let's break down how to use it in the following example:

```
import { Stream, Effect, Chunk, Option, StreamEmit } from "effect"

const events = [1, 2, 3, 4]

const stream = Stream.async(
  (emit: StreamEmit.Emit<never, never, number, void>) => {
    events.forEach((n) => {
      setTimeout(() => {
        if (n === 3) {
          emit(Effect.fail(Option.none())) // Terminate the stream
        } else {
          emit(Effect.succeed(Chunk.of(n))) // Add the current item to the stream
        }
      }, 100 * n)
    })
  }
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2 ]
}
*/
```

The `StreamEmit.Emit<R, E, A, void>` type represents an asynchronous callback that can be called multiple times. This callback takes a value of type `Effect<Chunk<A>, Option<E>, R>`. Here's what each of the possible outcomes means:

- When the value provided to the callback results in a `Chunk<A>` upon success, it signifies that the specified elements should be emitted as part of the stream.
- If the value passed to the callback results in a failure with `Some<E>`, it indicates the termination of the stream with the specified error.
- When the value passed to the callback results in a failure with `None`, it serves as a signal for the end of the stream, essentially terminating it.

To put it simply, this type allows you to specify how your asynchronous callback interacts with the stream, determining when to emit elements, when to terminate with an error, or when to signal the end of the stream.

From Iterables

fromIterable

You can create a pure stream from an `Iterable` of values using the `Stream.fromIterable` constructor. It's a straightforward way to convert a collection of values into a stream.

```
import { Stream } from "effect"

const numbers = [1, 2, 3]

const stream = Stream.fromIterable(numbers)
```

fromIterableEffect

When you have an effect that produces a value of type `Iterable`, you can employ the `Stream.fromIterableEffect` constructor to generate a stream from that effect.

For instance, let's say you have a database operation that retrieves a list of users. Since this operation involves effects, you can utilize `Stream.fromIterableEffect` to convert the result into a `Stream`:

```
import { Stream, Effect, Context } from "effect"

interface User {}

class Database extends Context.Tag("Database")<
  Database,
  { readonly getUsers: Effect.Effect<Array<User>> }
>() {}

const getUsers = Database.pipe(Effect.andThen((_) => _.getUsers))

const users = Stream.fromIterableEffect(getUsers)
```

This enables you to work seamlessly with effects and convert their results into streams for further processing.

fromAsyncIterable

Async iterables are another type of data source that can be converted into a stream. With the `Stream.fromAsyncIterable` constructor, you can work with asynchronous data sources and handle potential errors gracefully.

```
import { Stream } from "effect"

const myAsyncIterable = async function* () {
  yield 1
  yield 2
}

const stream = Stream.fromAsyncIterable(
  myAsyncIterable(),
  (e) => new Error(String(e)) // Error Handling
)
```

In this code, we define an async iterable and then create a stream named `stream` from it. Additionally, we provide an error handler function to manage any potential errors that may occur during the conversion.

From Repetition

Repeating a Single Value

You can create a stream that endlessly repeats a specific value using the `Stream.repeatValue` constructor:

```
import { Stream } from "effect"

const repeatZero = Stream.repeatValue(0)
```

Repeating a Stream's Content

`Stream.repeat` allows you to create a stream that repeats a specified stream's content according to a schedule. This can be useful for generating recurring events or values.

```
import { Stream, Schedule } from "effect"

// Creating a stream that repeats a value indefinitely
const repeatingStream = Stream.repeat(Stream.succeed(1), Schedule.forever)
```

Repeating an Effect's Result

Imagine you have an effectful API call, and you want to use the result of that call to create a stream. You can achieve this by creating a stream from the effect and repeating it indefinitely.

Here's an example of generating a stream of random numbers:

```
import { Stream, Random } from "effect"
```

```
const randomNumbers = Stream.repeatEffect(Random.nextInt)
```

Repeating an Effect with Termination

You can repeatedly evaluate a given effect and terminate the stream based on specific conditions.

In this example, we're draining an `Iterator` to create a stream from it:

```
import { Stream, Effect, Option } from "effect"

const drainIterator = <A>(it: Iterator<A>): Stream.Stream<A> =>
  Stream.repeatEffectOption(
    Effect.sync(() => it.next()).pipe(
      Effect.andThen((res) => {
        if (res.done) {
          return Effect.fail(Option.none())
        }
        return Effect.succeed(res.value)
      })
    )
  )
```

Generating Ticks

You can create a stream that emits `void` values at specified intervals using the `Stream.tick` constructor. This is useful for creating periodic events.

```
import { Stream } from "effect"

const stream = Stream.tick("2 seconds")
```

From Unfolding/Pagination

In functional programming, the concept of `unfold` can be thought of as the counterpart to `fold`.

With `fold`, we process a data structure and produce a return value. For example, we can take an `Array<number>` and calculate the sum of its elements.

On the other hand, `unfold` represents an operation where we start with an initial value and generate a recursive data structure, adding one element at a time using a specified state function. For example, we can create a sequence of natural numbers starting from `1` and using the `increment` function as the state function.

Unfold

unfold

The Stream module includes an `unfold` function defined as follows:

```
declare const unfold: <S, A>(
  initialState: S,
  step: (s: S) => Option.Option<readonly [A, S]>
) => Stream<A>
```

Here's how it works:

- **initialState**. This is the initial state value.
- **step**. The state function `step` takes the current state `s` as input. If the result of this function is `None`, the stream ends. If it's `Some<[A, S]>`, the next element in the stream is `A`, and the state `S` is updated for the next step process.

For example, let's create a stream of natural numbers using `Stream.unfold`:

```
import { Stream, Option } from "effect"

const nats = Stream.unfold(1, (n) => Option.some([n, n + 1]))
```

unfoldEffect

Sometimes, we may need to perform effectful state transformations during the unfolding operation. This is where `Stream.unfoldEffect` comes in handy. It allows us to work with effects while generating streams.

Here's an example of creating an infinite stream of random `1` and `-1` values using `Stream.unfoldEffect`:

```
import { Stream, Random, Effect, Option } from "effect"

const ints = Stream.unfoldEffect(1, (n) =>
  Random.nextBoolean.pipe(
    Effect.map((b) => (b ? Option.some([n, -n]) : Option.some([n, n])))
  )
)
```

Additional Variants

There are also similar operations like `Stream.unfoldChunk` and `Stream.unfoldChunkEffect` tailored for working with `Chunk` data types.

Pagination

paginate

`Stream.paginate` is similar to `Stream.unfold` but allows emitting values one step further.

For example, the following stream emits `0, 1, 2, 3` elements:

```
import { Stream, Option } from "effect"

const stream = Stream.paginate(0, (n) => [
  n,
  n < 3 ? Option.some(n + 1) : Option.none()
])
```

Here's how it works:

- We start with an initial value of `0`.
- The provided function takes the current value `n` and returns a tuple. The first element of the tuple is the value to emit (`n`), and the second element determines whether to continue (`Option.some(n + 1)`) or stop (`Option.none()`).

Additional Variants

There are also similar operations like `Stream.paginateChunk` and `Stream.paginateChunkEffect` tailored for working with `Chunk` data types.

Unfolding vs. Pagination

You might wonder about the difference between the `unfold` and `paginate` combinators and when to use one over the other. Let's explore this by diving into an example.

Imagine we have a paginated API that provides a substantial amount of data in a paginated manner. When we make a request to this API, it returns a `ResultPage` object containing the results for the current page and a flag indicating whether it's the last page or if there's more data to retrieve on the next page. Here's a simplified representation of our API:

```
import { Chunk, Effect } from "effect"

export type RawData = string

export class PageResult {
  constructor(
    readonly results: Chunk.Chunk<RawData>,
    readonly isLast: boolean
  ) {}
}

const pageSize = 2

export const listPaginated = (
  pageNumber: number
): Effect.Effect<PageResult, Error> => {
  return Effect.succeed(
    new PageResult(
      Chunk.map(
        Chunk.range(1, pageSize),
        (index) => `Result ${pageNumber}-${index}`
      ),
      pageNumber === 2 // Return 3 pages
    )
  )
}

// @include: domain
```

Our goal is to convert this paginated API into a stream of `RowData` events. For our initial attempt, we might think that using the `Stream.unfold` operation is the way to go:

```
// @filename: domain.ts
// @include: domain

// @filename: firstAttempt.ts
// ---cut---
import { Effect, Stream, Option } from "effect"
import { RawData, listPaginated } from "./domain"

const firstAttempt: Stream.Stream<RawData, Error> = Stream.unfoldChunkEffect(
  0,
  (pageNumber) =>
    listPaginated(pageNumber).pipe(
      Effect.map((page) => {
        if (page.isLast) {
          return Option.none()
        }
```

```
        return Option.some([page.results, pageNumber + 1] as const)
      })
    )
)

Effect.runPromise(Stream.runCollect(firstAttempt)).then(console.log)
/*
{
  _id: "Chunk",
  values: [ "Result 0-1", "Result 0-2", "Result 1-1", "Result 1-2" ]
}
*/
```

However, this approach has a drawback, it doesn't include the results from the last page. To work around this, we perform an extra API call to include those missing results:

```
// @filename: domain.ts
// @include: domain

// @filename: firstAttempt.ts
// ---cut---
import { Effect, Stream, Option } from "effect"
import { RawData, listPaginated } from "./domain"

const secondAttempt: Stream.Stream<RawData, Error> = Stream.unfoldChunkEffect(
  Option.some(0),
  (pageNumber) =>
    Option.match(pageNumber, {
      // We already hit the last page
      onNone: () => Effect.succeed(Option.none()),
      // We did not hit the last page yet
      onSome: (pageNumber) =>
        listPaginated(pageNumber).pipe(
          Effect.map((page) =>
            Option.some([
              page.results,
              page.isLast ? Option.none() : Option.some(pageNumber + 1)
            ])
          )
        )
    })
)

Effect.runPromise(Stream.runCollect(secondAttempt)).then(console.log)
/*
{
  _id: "Chunk",
  values: [ "Result 0-1", "Result 0-2", "Result 1-1", "Result 1-2", "Result 2-1", "Result 2-2" ]
}
*/
```

While this approach works, it's clear that `Stream.unfold` isn't the most friendly option for retrieving data from paginated APIs. It requires additional workarounds to include the results from the last page.

This is where `Stream.paginate` comes to the rescue. It provides a more ergonomic way to convert a paginated API into an Effect stream. Let's rewrite our solution using `Stream.paginate`:

```
// @filename: domain.ts
// @include: domain

// @filename: finalAttempt.ts
// ---cut---
import { Effect, Stream, Option } from "effect"
import { RawData, listPaginated } from "./domain"

const finalAttempt: Stream.Stream<RawData, Error> =
  Stream.paginateChunkEffect(0, (pageNumber) =>
    listPaginated(pageNumber).pipe(
      Effect.andThen((page) => {
        return [
          page.results,
          page.isLast ? Option.none<number>() : Option.some(pageNumber + 1)
        ]
      })
    )
  )

Effect.runPromise(Stream.runCollect(finalAttempt)).then(console.log)
/*
{
  _id: "Chunk",
  values: [ "Result 0-1", "Result 0-2", "Result 1-1", "Result 1-2", "Result 2-1", "Result 2-2" ]
}
*/
```

From Queue and PubSub

In Effect, there are two essential asynchronous messaging data types: Queue and PubSub. You can easily transform these data types into `Stream`s by utilizing `Stream.fromQueue` and `Stream.fromPubSub`, respectively.

From Schedule

We can create a stream from a `Schedule` that does not require any further input. The stream will emit an element for each value output from the schedule, continuing for as long as the schedule continues:

```
import { Stream, Schedule, Effect } from "effect"

// Emits values every 1 second for a total of 10 emissions
const schedule = Schedule.spaced("1 second").pipe(
  Schedule.compose(Schedule.recurs(10))
)

const stream = Stream.fromSchedule(schedule)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
}
*/
```

Operations

Location: 400-guides/660-streaming/100-stream/400-operations

In this section, explore essential stream operations, including tapping, taking elements, exploring streams as an alternative to async iterables, mapping, filtering, scanning, draining, detecting changes, zipping, grouping, concatenation, merging, interleaving, interspersing, broadcasting, buffering, and debouncing.

In this section, we'll explore some essential operations you can perform on streams. These operations allow you to manipulate and interact with stream elements in various ways.

Tapping

Tapping is an operation that involves running an effect on each emission of the stream. It allows you to observe each element, perform some effectful operation, and discard the result of this observation. Importantly, the `Stream.tap` operation does not alter the elements of the stream, and it does not affect the return type of the stream.

For instance, you can use `Stream.tap` to print each element of a stream:

```
import { Stream, Console, Effect } from "effect"

const stream = Stream.make(1, 2, 3).pipe(
  Stream.tap((n) => Console.log(`before mapping: ${n}`)),
  Stream.map((n) => n * 2),
  Stream.tap((n) => Console.log(`after mapping: ${n}`))
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
before mapping: 1
after mapping: 2
before mapping: 2
after mapping: 4
before mapping: 3
after mapping: 6
{
  _id: "Chunk",
  values: [ 2, 4, 6 ]
}
*/
```

Taking Elements

Another essential operation is taking elements, which allows you to extract a specific number of elements from a stream. Here are several ways to achieve this:

- `take`. To extract a fixed number of elements.
- `takeWhile`. To extract elements until a certain condition is met.
- `takeUntil`. To extract elements until a specific condition is met.
- `takeRight`. To extract a specified number of elements from the end.

```
import { Stream, Effect } from "effect"
```

```
const stream = Stream.iterate(0, (n) => n + 1)

// Using `take` to extract a fixed number of elements:
const s1 = Stream.take(stream, 5)
Effect.runPromise(Stream.runCollect(s1)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 0, 1, 2, 3, 4 ]
}
*/

// Using `takeWhile` to extract elements until a certain condition is met:
const s2 = Stream.takeWhile(stream, (n) => n < 5)
Effect.runPromise(Stream.runCollect(s2)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 0, 1, 2, 3, 4 ]
}
*/

// Using `takeUntil` to extract elements until a specific condition is met:
const s3 = Stream.takeUntil(stream, (n) => n === 5)
Effect.runPromise(Stream.runCollect(s3)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 0, 1, 2, 3, 4, 5 ]
}
*/

// Using `takeRight` to extract a specified number of elements from the end:
const s4 = Stream.takeRight(s3, 3)
Effect.runPromise(Stream.runCollect(s4)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 3, 4, 5 ]
}
*/
```

Exploring Streams as an Alternative to Async Iterables

When working with asynchronous data sources, like async iterables, you often need to consume data in a loop until a certain condition is met. This tutorial introduces how you can achieve similar behavior using Streams in a beginner-friendly manner.

In async iterables, data consumption can continue in a loop until a break or return statement is encountered. To replicate this behavior with Streams, you have a couple of options:

1. **Stream.takeUntil:** This function allows you to take elements from a stream until a specified condition evaluates to true. It's akin to breaking out of a loop in async iterables when a certain condition is met.

2. **Stream.toPull:** The `Stream.toPull` function is another way to replicate looping through async iterables. It returns an effect that repeatedly pulls data chunks from the stream. This effect can fail with `None` when the stream is finished or with `Some` error if it fails.

Let's take a closer look at the second option, `Stream.toPull`.

```
import { Stream, Effect } from "effect"

// Simulate a chunked stream
const stream = Stream.fromIterable([1, 2, 3, 4, 5]).pipe(Stream.rechunk(2))

const program = Effect.gen(function* () {
  // Create an effect to get data chunks from the stream
  const getChunk = yield* Stream.toPull(stream)

  // Continuously fetch and process chunks
  while (true) {
    const chunk = yield* getChunk
    console.log(chunk)
  }
})

Effect.runPromise(Effect.scoped(program)).then(console.log, console.error)
/*
Output:
{ _id: 'Chunk', values: [ 1, 2 ] }
{ _id: 'Chunk', values: [ 3, 4 ] }
{ _id: 'Chunk', values: [ 5 ] }
(FiberFailure) {
  "_id": "Option",
  "_tag": "None"
```

```
}
*/
```

In this example, we're using `Stream.toPull` to repeatedly pull data chunks from the `stream`. The code enters a loop and continues to fetch and display chunks until there's no more data left to process.

Mapping

In this section, we'll explore how to transform elements within a stream using the `Stream.map` family of operations. These operations allow you to apply a function to each element of the stream, producing a new stream with the transformed values.

Basic Mapping

The `Stream.map` operation applies a given function to all elements of the stream, creating another stream with the transformed values. Let's illustrate this with an example:

```
import { Stream, Effect } from "effect"

const stream = Stream.make(1, 2, 3).pipe(Stream.map((n) => n + 1))

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 2, 3, 4 ]
}
*/
```

Effectful Mapping

If your transformation involves effects, you can use `Stream.mapEffect` instead. It allows you to apply an effectful function to each element of the stream, producing a new stream with effectful results:

```
import { Stream, Random, Effect } from "effect"

const stream = Stream.make(10, 20, 30).pipe(
  Stream.mapEffect((n) => Random.nextIntBetween(0, n))
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 6, 13, 5 ]
}
*/
```

You can evaluate effects concurrently using the `concurrency` option. It allows you to specify the number of concurrent running effects. The results are emitted downstream in the original order.

Let's write a simple page downloader that fetches URLs concurrently:

```
import { Stream, Effect } from "effect"

const getUrls = Effect.succeed(["url0", "url1", "url2"])

const fetchUrl = (url: string) =>
  Effect.succeed([
    `Resource 0-${url}`,
    `Resource 1-${url}`,
    `Resource 2-${url}`
  ])

const stream = Stream.fromIterableEffect(getUrls).pipe(
  Stream.mapEffect(fetchUrl, { concurrency: 4 })
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ "Resource 0-url0", "Resource 1-url0", "Resource 2-url0" ], [ "Resource 0-url1", "Resource 1-url1",
      "Resource 2-url1" ], [ "Resource 0-url2", "Resource 1-url2", "Resource 2-url2" ]
  ]
}
*/
```

Stateful Mapping

The `Stream.mapAccum` operation is similar to `Stream.map`, but it transforms elements statefully and allows you to map and accumulate in a single operation. Let's see how you can use it to calculate the running total of an input stream:

```
import { Stream, Effect } from "effect"

const runningTotal = (stream: Stream.Stream<number>): Stream.Stream<number> =>
  stream.pipe(Stream.mapAccum(0, (s, a) => [s + a, s + a]))

// input:  0, 1, 2, 3, 4, 5
Effect.runPromise(Stream.runCollect(runningTotal(Stream.range(0, 6)))).then(
  console.log
)
/*
Output:
{
  _id: "Chunk",
  values: [ 0, 1, 3, 6, 10, 15 ]
}
*/
```

Mapping and Flattening

The `Stream.mapConcat` operation is akin to `Stream.map`, but it takes things a step further. It maps each element to zero or more elements of type `Iterable` and then flattens the entire stream. Let's illustrate this with an example:

```
import { Stream, Effect } from "effect"

const numbers = Stream.make("1-2-3", "4-5", "6").pipe(
  Stream.mapConcat((s) => s.split("-")),
  Stream.map((s) => parseInt(s))
)

Effect.runPromise(Stream.runCollect(numbers)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, 4, 5, 6 ]
}
*/
```

In this example, we take a stream of strings like `"1-2-3"` and split them into individual numbers, resulting in a flattened stream of integers.

Mapping to a Constant Value

The `Stream.as` method allows you to map the success values of a stream to a specified constant value. This can be handy when you want to transform elements into a uniform value. Here's an example where we map all elements to the `null` value:

```
import { Stream, Effect } from "effect"

const stream = Stream.range(1, 5).pipe(Stream.as(null))

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ null, null, null, null ]
}
*/
```

In this case, regardless of the original values in the stream, we've mapped them all to `null`.

Filtering

The `Stream.filter` operation is like a sieve that lets through elements that meet a specified condition. Think of it as a way to sift through a stream and keep only the elements that satisfy the given criteria. Here's an example:

```
import { Stream, Effect } from "effect"

const stream = Stream.range(1, 11).pipe(Stream.filter((n) => n % 2 === 0))

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 2, 4, 6, 8, 10 ]
}
*/
```

In this example, we start with a stream of numbers from 1 to 10 and use `Stream.filter` to retain only the even numbers (those that satisfy the condition `n % 2 === 0`). The result is a filtered stream containing the even numbers from the original stream.

Scanning

In this section, we'll explore the concept of stream scanning. Scans are similar to folds, but they provide a historical perspective. Like folds, scans also involve a binary operator and an initial value. However, what makes scans unique is that they emit every intermediate result as part of the stream.

```
import { Stream, Effect } from "effect"

const stream = Stream.range(1, 6).pipe(Stream.scan(0, (a, b) => a + b))

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 0, 1, 3, 6, 10, 15 ]
}
*/
```

In this example, we have a stream of numbers from 1 to 5, and we use `Stream.scan` to perform a cumulative addition starting from an initial value of 0. The result is a stream that emits the accumulated sum at each step: 0, 1, 3, 6, 10, and 15.

Streams scans provide a way to keep a historical record of your stream transformations, which can be invaluable for various applications.

Additionally, if you only need the final result of the scan, you can use `Stream.runFold`:

```
import { Stream, Effect } from "effect"

const fold = Stream.range(1, 6).pipe(Stream.runFold(0, (a, b) => a + b))

Effect.runPromise(fold).then(console.log) // Output: 15
```

In this case, `Stream.runFold` gives you the final accumulated value, which is 15 in this example.

Draining

In this section, we'll explore the concept of stream draining. Imagine you have a stream filled with effectful operations, but you're not interested in the values they produce; instead, you want to execute these effects and discard the results. This is where the `Stream.drain` function comes into play.

Let's go through a few examples:

Example 1: Discarding Values

```
import { Stream, Effect } from "effect"

// We create a stream and immediately drain it.
const s1 = Stream.range(1, 6).pipe(Stream.drain)

Effect.runPromise(Stream.runCollect(s1)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: []
}
*/
```

In this example, we have a stream with values from 1 to 5, but we use `Stream.drain` to discard these values. As a result, the output stream is empty.

Example 2: Executing Random Effects

```
import { Stream, Effect, Random } from "effect"

const s2 = Stream.repeatEffect(
  Effect.gen(function* () {
    const nextInt = yield* Random.nextInt
    const number = Math.abs(nextInt % 10)
    console.log(`random number: ${number}`)
    return number
  })
).pipe(Stream.take(3))

Effect.runPromise(Stream.runCollect(s2)).then(console.log)
/*
Output:
random number: 4
random number: 2
random number: 7
{
  _id: "Chunk",
  values: [ 4, 2, 7 ]
}
*/
```

```
const s3 = Stream.drain(s2)

Effect.runPromise(Stream.runCollect(s3)).then(console.log)
/*
random number: 1
random number: 6
random number: 0
Output:
{
  _id: "Chunk",
  values: []
}
*/
```

In this example, we create a stream with random effects and collect the values of these effects initially. Later, we use `Stream.drain` to execute the same effects without collecting the values. This demonstrates how you can use draining to trigger effectful operations when you're not interested in the emitted values.

Stream draining can be particularly useful when you need to perform certain actions or cleanup tasks in your application without affecting the main stream of data.

Detecting Changes in a Stream

In this section, we'll explore the `Stream.changes` operation, which allows you to detect and emit elements that are different from their preceding elements within the stream.

```
import { Stream, Effect } from "effect"

const stream = Stream.make(1, 1, 1, 2, 2, 3, 4).pipe(Stream.changes)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, 4 ]
}
*/
```

Zipping

Zipping is a process of combining two or more streams to create a new stream by pairing elements from the input streams. We can achieve this using the `Stream.zip` and `Stream.zipWith` operators. Let's dive into some examples:

```
import { Stream, Effect } from "effect"

// We create two streams and zip them together.
const s1 = Stream.zip(
  Stream.make(1, 2, 3, 4, 5, 6),
  Stream.make("a", "b", "c")
)

Effect.runPromise(Stream.runCollect(s1)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ 1, "a" ], [ 2, "b" ], [ 3, "c" ]
  ]
}
*/

// We create two streams and zip them with custom logic.
const s2 = Stream.zipWith(
  Stream.make(1, 2, 3, 4, 5, 6),
  Stream.make("a", "b", "c"),
  (n, s) => [n - s.length, s]
)

Effect.runPromise(Stream.runCollect(s2)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ 0, "a" ], [ 1, "b" ], [ 2, "c" ]
  ]
}
*/
```

The new stream will end when one of the streams ends.

Handling Stream Endings

When one of the input streams ends before the other, you might need to zip with default values. The `Stream.zipAll` and `Stream.zipAllWith` operations allow you to specify default values for both sides to handle such scenarios. Let's see an example:

```
import { Stream, Effect } from "effect"

const s1 = Stream.zipAll(Stream.make(1, 2, 3, 4, 5, 6), {
  other: Stream.make("a", "b", "c"),
  defaultSelf: 0,
  defaultOther: "x"
})

Effect.runPromise(Stream.runCollect(s1)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ 1, "a" ], [ 2, "b" ], [ 3, "c" ], [ 4, "x" ], [ 5, "x" ], [ 6, "x" ]
  ]
}
*/

const s2 = Stream.zipAllWith(Stream.make(1, 2, 3, 4, 5, 6), {
  other: Stream.make("a", "b", "c"),
  onSelf: (n) => [n, "x"],
  onOther: (s) => [0, s],
  onBoth: (n, s) => [n - s.length, s]
})

Effect.runPromise(Stream.runCollect(s2)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ 0, "a" ], [ 1, "b" ], [ 2, "c" ], [ 4, "x" ], [ 5, "x" ], [ 6, "x" ]
  ]
}
*/
```

This allows you to handle zipping when one stream completes earlier than the other.

Zipping Streams at Different Rates

Sometimes, you might have two streams producing elements at different speeds. If you don't want to wait for the slower one when zipping elements, you can use `Stream.zipLatest` or `Stream.zipLatestWith`. These operations combine elements in a way that when a value is emitted by either of the two streams, it is combined with the latest value from the other stream to produce a result. Here's an example:

```
import { Stream, Schedule, Effect } from "effect"

const s1 = Stream.make(1, 2, 3).pipe(
  Stream.schedule(Schedule.spaced("1 second"))
)

const s2 = Stream.make("a", "b", "c", "d").pipe(
  Stream.schedule(Schedule.spaced("500 millis"))
)

const stream = Stream.zipLatest(s1, s2)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ 1, "a" ], [ 1, "b" ], [ 2, "b" ], [ 2, "c" ], [ 2, "d" ], [ 3, "d" ]
  ]
}
*/
```

Here, `Stream.zipLatest` combines elements from both streams without waiting for the slower one, resulting in a more responsive output.

Pairing with Previous and Next Elements

- `zipWithPrevious`: This operator pairs each element of a stream with its previous element.
- `zipWithNext`: It pairs each element of a stream with its next element.
- `zipWithPreviousAndNext`: This operator pairs each element with both its previous and next elements.

Here's an example illustrating these operations:

```
import { Stream } from "effect"

const stream = Stream.make(1, 2, 3, 4)
```

```
const s1 = Stream.zipWithPrevious(stream)

const s2 = Stream.zipWithNext(stream)

const s3 = Stream.zipWithPreviousAndNext(stream)
```

Indexing Stream Elements

Another handy operator is `Stream.zipWithIndex`, which indexes each element of a stream by pairing it with its respective index. This is especially useful when you need to keep track of the position of elements within the stream.

Here's an example of indexing elements in a stream:

```
import { Stream, Effect } from "effect"

const stream = Stream.make("Mary", "James", "Robert", "Patricia")

const indexedStream = Stream.zipWithIndex(stream)

Effect.runPromise(Stream.runCollect(indexedStream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ "Mary", 0 ], [ "James", 1 ], [ "Robert", 2 ], [ "Patricia", 3 ]
  ]
}
*/
```

Cartesian Product of Streams

The Stream module introduces a powerful feature: the ability to compute the *Cartesian Product* of two streams. This operation allows you to generate combinations of elements from two separate streams. Let's explore this concept further:

Imagine you have two sets of items, and you want to generate all possible pairs by taking one item from each set. This process is known as finding the Cartesian Product of the sets. In the context of streams, it means creating combinations of elements from two streams.

To achieve this, the Stream module provides the `Stream.cross` operator, along with its variants. These operators take two streams and generate a new stream containing all possible combinations of elements from the original streams.

Here's a practical example:

```
import { Stream, Effect } from "effect"

const s1 = Stream.make(1, 2, 3)
const s2 = Stream.make("a", "b")

const product = Stream.cross(s1, s2)

Effect.runPromise(Stream.runCollect(product)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ 1, "a" ], [ 1, "b" ], [ 2, "a" ], [ 2, "b" ], [ 3, "a" ], [ 3, "b" ]
  ]
}
*/
```

It's important to note that the right-hand side stream (`s2` in this case) will be iterated multiple times, once for each element in the left-hand side stream (`s1`). This means that if the right-hand side stream involves expensive or side-effectful operations, they will be executed multiple times.

Partitioning

Partitioning a stream means dividing it into two separate streams based on a specified condition. The Stream module provides two helpful functions for achieving this: `Stream.partition` and `Stream.partitionEither`. Let's explore how these functions work and when to use them.

partition

The `Stream.partition` function takes a predicate as input and splits the original stream into two substreams: one containing elements that satisfy the predicate (evaluate to `true`), and the other containing elements that do not (evaluate to `false`). Additionally, these substreams are wrapped within a `Scope` type.

Here's an example where we partition a stream of numbers into even and odd numbers:

```
import { Stream, Effect } from "effect"

const partition = Stream.range(1, 10).pipe(
  Stream.partition((n) => n % 2 === 0, { bufferSize: 5 })
```

```
)

Effect.runPromise(
  Effect.scoped(
    Effect.gen(function* () {
      const [evens, odds] = yield* partition
      console.log(yield* Stream.runCollect(evens))
      console.log(yield* Stream.runCollect(odds))
    })
  )
)
/*
Output:
{
  _id: "Chunk",
  values: [ 2, 4, 6, 8 ]
}
{
  _id: "Chunk",
  values: [ 1, 3, 5, 7, 9 ]
}
*/
```

In this example, we use the `Stream.partition` function with a predicate to split the stream into even and odd numbers. The `bufferSize` option controls how much the faster stream can advance beyond the slower one.

partitionEither

Sometimes, you may need to partition a stream using an effectful predicate. For such cases, the `Stream.partitionEither` function is available. This function accepts an effectful predicate and divides the stream into two substreams based on the result of the predicate: elements that yield `Either.left` values go to one substream, while elements yielding `Either.right` values go to the other.

Here's an example where we use `Stream.partitionEither` to partition a stream of numbers based on an effectful condition:

```
import { Stream, Effect, Either } from "effect"

const partition = Stream.range(1, 10).pipe(
  Stream.partitionEither(
    (n) => Effect.succeed(n < 5 ? Either.left(n * 2) : Either.right(n)),
    { bufferSize: 5 }
  )
)

Effect.runPromise(
  Effect.scoped(
    Effect.gen(function* () {
      const [evens, odds] = yield* partition
      console.log(yield* Stream.runCollect(evens))
      console.log(yield* Stream.runCollect(odds))
    })
  )
)
/*
Output:
{
  _id: "Chunk",
  values: [ 2, 4, 6, 8 ]
}
{
  _id: "Chunk",
  values: [ 5, 6, 7, 8, 9 ]
}
*/
```

In this case, the `Stream.partitionEither` function splits the stream into two substreams: one containing values that are less than 5 (doubled using `Either.left`), and the other containing values greater than or equal to 5 (using `Either.right`).

GroupBy

When working with streams of data, you may often need to group elements based on certain criteria. The Stream module provides two functions for achieving this: `groupByKey` and `groupBy`. Let's explore how these functions work and when to use them.

groupByKey

The `Stream.groupByKey` function allows you to partition a stream by a simple function of type `(a: A) => K`, where `A` represents the type of elements in your stream, and `K` represents the keys by which the stream should be partitioned. This function is not effectful, it simply groups elements by applying the provided function.

The `Stream.groupByKey` function returns a new data type called `GroupBy`. This `GroupBy` type represents a grouped stream. To work with the groups, you can use the `GroupBy.evaluate` function, which takes a function of type `(key: K, stream: Stream<V, E>) => Stream.Stream<...>`. This function runs across all groups and merges them in a non-deterministic fashion.

In the example below, we use `groupByKey` to group exam results by the tens place of their scores and count the number of results in each group:

```
import { Stream, GroupBy, Effect, Chunk } from "effect"

class Exam {
  constructor(
    readonly person: string,
    readonly score: number
  ) {}
}

const examResults = [
  new Exam("Alex", 64),
  new Exam("Michael", 97),
  new Exam("Bill", 77),
  new Exam("John", 78),
  new Exam("Bobby", 71)
]

const groupByKeyResult = Stream.fromIterable(examResults).pipe(
  Stream.groupByKey((exam) => Math.floor(exam.score / 10) * 10)
)

const stream = GroupBy.evaluate(groupByKeyResult, (key, stream) =>
  Stream.fromEffect(
    Stream.runCollect(stream).pipe(
      Effect.andThen((chunk) => [key, Chunk.size(chunk)] as const)
    )
  )
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ 60, 1 ], [ 90, 1 ], [ 70, 3 ]
  ]
}
*/
```

In this example, we partition the exam results into groups based on the tens place of their scores (e.g., 60, 90, 70). The `groupByKey` function is ideal for simple, non-effectful partitioning.

groupBy

In more complex scenarios where partitioning involves effects, you can turn to the `Stream.groupBy` function. This function takes an effectful partitioning function and generates a `GroupBy` data type, representing a grouped stream. You can then use `GroupBy.evaluate` in a similar fashion as before to process the groups.

In the following example, we group names by their first character and count the number of names in each group. Note that the partitioning operation itself is simulated as effectful:

```
import { Stream, GroupBy, Effect, Chunk } from "effect"

const groupByKeyResult = Stream.fromIterable([
  "Mary",
  "James",
  "Robert",
  "Patricia",
  "John",
  "Jennifer",
  "Rebecca",
  "Peter"
]).pipe(
  Stream.groupBy((name) => Effect.succeed([name.substring(0, 1), name]))
)

const stream = GroupBy.evaluate(groupByKeyResult, (key, stream) =>
  Stream.fromEffect(
    Stream.runCollect(stream).pipe(
      Effect.andThen((chunk) => [key, Chunk.size(chunk)] as const)
    )
  )
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    [ "M", 1 ], [ "J", 3 ], [ "R", 2 ], [ "P", 2 ]
  ]
}
*/
```

Grouping

When working with streams, you may encounter situations where you need to group elements in a more structured manner. The Stream module provides two helpful functions for achieving this: `grouped` and `groupedWithin`. In this section, we'll explore how these functions work and when to use them.

grouped

The `Stream.grouped` function is perfect for partitioning stream results into chunks of a specified size. It's especially useful when you want to work with data in smaller, more manageable pieces.

Here's an example that demonstrates the use of `Stream.grouped`:

```
import { Stream, Effect } from "effect"

const stream = Stream.range(0, 8).pipe(Stream.grouped(3))

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    {
      _id: "Chunk",
      values: [ 0, 1, 2 ]
    }, {
      _id: "Chunk",
      values: [ 3, 4, 5 ]
    }, {
      _id: "Chunk",
      values: [ 6, 7 ]
    }
  ]
}
*/
```

In this example, we take a stream of numbers from 0 to 9 and use `Stream.grouped(3)` to divide it into chunks of size 3.

groupedWithin

The `Stream.groupedWithin` function provides more flexibility by allowing you to group events based on time intervals or chunk size, whichever condition is satisfied first. This is particularly useful when you want to group data based on time constraints.

```
import { Stream, Schedule, Effect } from "effect"

const stream = Stream.range(0, 10).pipe(
  Stream.repeat(Schedule.spaced("1 second")),
  Stream.groupedWithin(18, "1.5 seconds"),
  Stream.take(3)
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [
    {
      _id: "Chunk",
      values: [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7 ]
    }, {
      _id: "Chunk",
      values: [ 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
    }, {
      _id: "Chunk",
      values: [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7 ]
    }
  ]
}
*/
```

In this example, we use `Stream.groupedWithin(18, "1.5 seconds")` to create chunks of data. The grouping operation occurs either when 18 elements are reached or when 1.5 seconds have passed since the last chunk was created. This is particularly useful when dealing with time-sensitive data or when you want to control the chunk size dynamically.

Concatenation

In stream processing, there are scenarios where you may want to combine the contents of multiple streams. The Stream module provides several operators for achieving this, including `Stream.concat`, `Stream.concatAll`, and `Stream.flatMap`. Let's explore these operators and understand how to use them effectively.

Simple Concatenation

The `Stream.concat` operator is a straightforward way to concatenate two streams. It returns a new stream that emits elements from the left-hand stream followed by elements from the right-hand stream. This is useful when you want to combine two streams in a sequential manner.

Here's an example of using `Stream.concat`:

```
import { Stream, Effect } from "effect"

const s1 = Stream.make(1, 2, 3)
const s2 = Stream.make(4, 5)

const stream = Stream.concat(s1, s2)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, 4, 5 ]
}
*/
```

Concatenating Multiple Streams

Sometimes you may have multiple streams that you want to concatenate together. Instead of manually chaining `Stream.concat` operations, you can use `Stream.concatAll` to concatenate a `Chunk` of streams.

Here's an example:

```
import { Stream, Effect, Chunk } from "effect"

const s1 = Stream.make(1, 2, 3)
const s2 = Stream.make(4, 5)
const s3 = Stream.make(6, 7, 8)

const stream = Stream.concatAll(Chunk.make(s1, s2, s3))

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 3, 4, 5, 6, 7, 8 ]
}
*/
```

Advanced Concatenation with flatMap

The `Stream.flatMap` operator allows you to create a stream whose elements are generated by applying a function of type `(a: A) => Stream<...>` to each output of the source stream. It concatenates all of the results.

Here's an example of using `Stream.flatMap`:

```
import { Stream, Effect } from "effect"

const stream = Stream.make(1, 2, 3).pipe(
  Stream.flatMap((a) => Stream.repeatValue(a).pipe(Stream.take(4)))
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3 ]
}
*/
```

If we need to do the `flatMap` concurrently, we can use the `concurrency` option, and also if the order of concatenation is not important for us, we can use the `switch` option.

Merging

Sometimes we need to interleave the emission of two streams and create another stream. In these cases, we can't use the `Stream.concat` operation because the concat operation waits for the first stream to finish and then consumes the second stream. So we need a way of picking elements from different sources. Effect Stream's merge operations does this for us. Let's discuss some variants of this operation:

merge

The `Stream.merge` operation allows us to pick elements from different source streams and merge them into a single stream. Unlike `Stream.concat`, which waits for the first stream to finish before moving to the second, `Stream.merge` interleaves elements from both streams as they become available.

Here's an example:

```
import { Schedule, Stream, Effect } from "effect"

const s1 = Stream.make(1, 2, 3).pipe(
  Stream.schedule(Schedule.spaced("100 millis"))
)
const s2 = Stream.make(4, 5, 6).pipe(
  Stream.schedule(Schedule.spaced("200 millis"))
)

const stream = Stream.merge(s1, s2)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 4, 2, 3, 5, 6 ]
}
*/
```

Termination Strategy

When merging two streams, we should consider their termination strategy. Each stream has its own lifetime, some may finish quickly, while others may continue indefinitely. By default, when using `Stream.merge`, the resulting stream terminates only when both specified streams terminate.

However, you can define the termination strategy to align with your requirements. Stream offers four different termination strategies using the `haltStrategy` option:

- `"left"`. The resulting stream will terminate when the left-hand side stream terminates.
- `"right"`. The resulting stream will terminate when the right-hand side stream finishes.
- `"both"`. The resulting stream will terminate when both streams finish.
- `"either"`. The resulting stream will terminate when one of the streams finishes.

Here's an example of specifying a termination strategy:

```
import { Stream, Schedule, Effect } from "effect"

const s1 = Stream.range(1, 6).pipe(
  Stream.schedule(Schedule.spaced("100 millis"))
)
const s2 = Stream.repeatValue(0).pipe(
  Stream.schedule(Schedule.spaced("200 millis"))
)

const stream = Stream.merge(s1, s2, { haltStrategy: "left" })

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 0, 2, 3, 0, 4, 5 ]
}
*/
```

In this example, we use `haltStrategy: "left"` to make the resulting stream terminate when the left-hand stream (`s1`) finishes.

mergeWith

In some cases, we not only want to merge two streams but also transform and unify their elements into new types. This is where `Stream.mergeWith` comes into play. It allows us to specify transformation functions for both source streams.

Here's an example:

```
import { Schedule, Stream, Effect } from "effect"

const s1 = Stream.make("1", "2", "3").pipe(
  Stream.schedule(Schedule.spaced("100 millis"))
)
const s2 = Stream.make(4.1, 5.3, 6.2).pipe(
  Stream.schedule(Schedule.spaced("200 millis"))
)

const stream = Stream.mergeWith(s1, s2, {
  onSelf: (s) => parseInt(s),
  onOther: (n) => Math.floor(n)
})

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 4, 2, 3, 5, 6 ]
```

```
}
*/
```

In this example, we use `Stream.mergeWith` to merge `s1` and `s2` while converting string elements from `s1` to integers and rounding decimal elements from `s2`.

Interleaving

The `Stream.interleave` operator allows us to pull one element at a time from each of two streams, creating a new interleaved stream. Once one of the streams is exhausted, the remaining values from the other stream are pulled.

Here's an example:

```
import { Stream, Effect } from "effect"

const s1 = Stream.make(1, 2, 3)
const s2 = Stream.make(4, 5, 6)

const stream = Stream.interleave(s1, s2)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 4, 2, 5, 3, 6 ]
}
*/
```

For more advanced interleaving logic, `Stream.interleaveWith` provides additional flexibility. It allows you to specify the interleaving logic using a third stream of `boolean` values. When the boolean stream emits `true`, it chooses elements from the left-hand stream; otherwise, it selects elements from the right-hand stream.

Here's an example:

```
import { Stream, Effect } from "effect"

const s1 = Stream.make(1, 3, 5, 7, 9)
const s2 = Stream.make(2, 4, 6, 8, 10)

const booleanStream = Stream.make(true, false, false).pipe(Stream.forever)

const stream = Stream.interleaveWith(s1, s2, booleanStream)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 2, 4, 3, 6, 8, 5, 10, 7, 9 ]
}
*/
```

In this example, `booleanStream` decides which source stream to choose for interleaving. When `true`, it picks elements from `s1`, and when `false`, it selects elements from `s2`.

Interspersing

Interspersing is a technique that allows you to add separators in a stream. This can be especially useful when you want to format or structure the data in your streams.

intersperse

The `Stream.intersperse` operator lets you intersperse a delimiter element between the elements of a stream. This delimiter can be any value you choose. It's added between each pair of elements in the original stream.

Here's an example:

```
import { Stream, Effect } from "effect"

const stream = Stream.make(1, 2, 3, 4, 5).pipe(Stream.intersperse(0))

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 1, 0, 2, 0, 3, 0, 4, 0, 5 ]
}
*/
```

In this example, we have a stream `stream` with numbers from 1 to 5, and we use `Stream.intersperse(0)` to add zeros between them.

intersperseAffixes

For more advanced interspersing needs, `Stream.intersperseAffixes` provides greater control. It allows you to specify different affixes for the start, middle, and end of your stream. These affixes can be strings or any other values you want.

Here's an example:

```
import { Stream, Effect } from "effect"

const stream = Stream.make(1, 2, 3, 4, 5).pipe(
  Stream.intersperseAffixes({
    start: "[",
    middle: "-",
    end: "]"
  })
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ "[", 1, "-", 2, "-", 3, "-", 4, "-", 5, "]" ]
}
*/
```

In this example, we use `Stream.intersperseAffixes` to enclose the numbers from 1 to 5 within square brackets, separating them with hyphens.

Broadcasting

Broadcasting a stream is a way to create multiple streams that contain the same elements as the source stream. This operation allows you to send each element to multiple downstream streams simultaneously. However, the upstream stream can emit events only up to a certain limit, which is determined by the `maximumLag` parameter. Once this limit is reached, the upstream stream slows down to match the speed of the slowest downstream stream.

Let's take a closer look at how broadcasting works in the following example. Here, we are broadcasting a stream of numbers to two downstream streams. One of them calculates the maximum number in the stream, while the other performs some logging with an additional delay. The upstream stream adjusts its speed based on the slower logging stream:

```
import { Effect, Stream, Console, Schedule, Fiber } from "effect"

const numbers = Effect.scoped(
  Stream.range(1, 21).pipe(
    Stream.tap((n) => Console.log(`Emit ${n} element before broadcasting`)),
    Stream.broadcast(2, 5),
    Stream.flatMap(([first, second]) =>
      Effect.gen(function* () {
        const fiber1 = yield* Stream.runFold(first, 0, (acc, e) =>
          Math.max(acc, e)
        ).pipe(
          Effect.andThen((max) => Console.log(`Maximum: ${max}`)),
          Effect.fork
        )
        const fiber2 = yield* second.pipe(
          Stream.schedule(Schedule.spaced("1 second")),
          Stream.runForEach((n) =>
            Console.log(`Logging to the Console: ${n}`)
          ),
          Effect.fork
        )
        yield* Fiber.join(fiber1).pipe(
          Effect.zip(Fiber.join(fiber2), { concurrent: true })
        )
      })
    ),
    Stream.runCollect
  )
)

Effect.runPromise(numbers).then(console.log)
/*
Output:
Emit 1 element before broadcasting
Emit 2 element before broadcasting
Emit 3 element before broadcasting
Emit 4 element before broadcasting
Emit 5 element before broadcasting
Emit 6 element before broadcasting
Emit 7 element before broadcasting
Emit 8 element before broadcasting
Emit 9 element before broadcasting
Emit 10 element before broadcasting
Emit 11 element before broadcasting
Logging to the Console: 1
Logging to the Console: 2
Logging to the Console: 3
```

```
Logging to the Console: 4
Logging to the Console: 5
Emit 12 element before broadcasting
Emit 13 element before broadcasting
Emit 14 element before broadcasting
Emit 15 element before broadcasting
Emit 16 element before broadcasting
Logging to the Console: 6
Logging to the Console: 7
Logging to the Console: 8
Logging to the Console: 9
Logging to the Console: 10
Emit 17 element before broadcasting
Emit 18 element before broadcasting
Emit 19 element before broadcasting
Emit 20 element before broadcasting
Logging to the Console: 11
Logging to the Console: 12
Logging to the Console: 13
Logging to the Console: 14
Logging to the Console: 15
Maximum: 20
Logging to the Console: 16
Logging to the Console: 17
Logging to the Console: 18
Logging to the Console: 19
Logging to the Console: 20
{
  _id: "Chunk",
  values: [ undefined ]
}
*/
```

Buffering

Effect streams operate in a pull-based manner, which means downstream consumers can request elements at their own pace without needing to signal the upstream to slow down. However, there are scenarios where you might need to handle producers and consumers independently, especially when there's a speed mismatch between them. This is where buffering comes into play, allowing you to manage communication between a faster producer and a slower consumer effectively. Effect streams provide a built-in `Stream.buffer` operator to assist with this.

buffer

The `Stream.buffer` operator is designed to facilitate scenarios where a faster producer needs to work independently of a slower consumer. It achieves this by buffering elements in a queue, allowing the producer to continue working even if the consumer lags behind. You can specify the maximum buffer capacity using the `capacity` option.

Let's walk through an example to see how it works:

```
import { Stream, Console, Schedule, Effect } from "effect"

const stream = Stream.range(1, 11).pipe(
  Stream.tap((n) => Console.log(`before buffering: ${n}`)),
  Stream.buffer({ capacity: 4 }),
  Stream.tap((n) => Console.log(`after buffering: ${n}`)),
  Stream.schedule(Schedule.spaced("5 seconds"))
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
before buffering: 1
before buffering: 2
before buffering: 3
before buffering: 4
before buffering: 5
before buffering: 6
after buffering: 1
after buffering: 2
before buffering: 7
after buffering: 3
before buffering: 8
after buffering: 4
before buffering: 9
after buffering: 5
before buffering: 10
...
*/
```

In this example, we create a stream of numbers from 1 to 11. We use `Stream.buffer({ capacity: 4 })` to buffer up to 4 elements at a time. As you can see, the `Stream.tap` operator allows us to log each element before and after buffering. We've also introduced a 5-second delay between each emission to illustrate the lag between producing and consuming messages.

You can choose from different buffering options based on the type of underlying queue you need:

- Bounded Queue: `{ capacity: number }`

- Unbounded Queue: `{ capacity: "unbounded" }`
- Sliding Queue: `{ capacity: number, strategy: "sliding" }`
- Dropping Queue: `{ capacity: number, strategy: "dropping" }`

Debouncing

The `Stream.debounce` function plays a crucial role in controlling the rate at which elements are emitted. It introduces a minimum time interval between the emission of each element. This ensures that elements are emitted at a more controlled pace, especially when dealing with rapid or frequent emissions.

```
import { Stream, Effect } from "effect"

const stream = Stream.make(1, 2, 3).pipe(
  Stream.concat(Stream.fromEffect(Effect.sleep("500 millis"))),
  Stream.concat(Stream.make(4, 5)),
  Stream.concat(Stream.fromEffect(Effect.sleep("10 millis"))),
  Stream.concat(Stream.make(6)),
  Stream.debounce("100 millis") // Emit only after a pause of at least 100 ms
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 3, 6 ]
}
*/
```

In this example, we have a stream that emits elements at varying intervals. Some elements are emitted rapidly, while others are separated by pauses of different durations. We apply debouncing with a minimum pause requirement of 100 milliseconds using `Stream.debounce("100 millis")`.

The result is that only elements that follow a pause of at least 100 milliseconds are emitted. This means that elements 1, 2, 4, and 5 are effectively skipped because they are emitted too close together. Only elements 3 and 6, which have a pause of at least 100 milliseconds before them, are emitted.

Introduction to Streams

Location: 400-guides/660-streaming/100-stream/100-introduction

Discover the power of `Stream` in Effect, a program description that goes beyond the capabilities of `Effect`. Unlike an `Effect` that always produces a single result, a `Stream` can emit zero or more values of type `A`, making it a versatile tool for various tasks. Explore scenarios such as empty streams, single-element streams, finite streams, and infinite streams. Learn how to use `Stream` to handle a wide range of tasks, from processing finite lists to dealing with infinite sequences.

In this guide, we'll explore the concept of a `Stream<A, E, R>`. A `Stream` is a program description that, when executed, can emit **zero or more values** of type `A`, handle errors of type `E`, and operates within a context of type `R`.

Use Cases

Streams are particularly handy whenever you're dealing with sequences of values over time. They can serve as replacements for observables, node streams, and AsyncIterables.

What is a Stream?

Think of a `Stream` as an extension of an `Effect`. While an `Effect<A, E, R>` represents a program that requires a context of type `R`, may encounter an error of type `E`, and always produces a single result of type `A`, a `Stream<A, E, R>` takes this further by allowing the emission of zero or more values of type `A`.

To clarify, let's examine some examples using `Effect`:

```
import { Effect, Chunk, Option } from "effect"

// An Effect that fails with a string error
const failedEffect = Effect.fail("fail!")

// An Effect that produces a single number
const oneNumberValue = Effect.succeed(3)

// An Effect that produces a chunk of numbers
const oneListValue = Effect.succeed(Chunk.make(1, 2, 3))

// An Effect that produces an optional number
const oneOption = Effect.succeed(Option.some(1))
```

In each case, the `Effect` always ends with **exactly one value**. There is no variability; you always get one result.

Understanding Streams

Now, let's shift our focus to `Stream`. A `Stream` represents a program description that shares similarities with `Effect`, it requires a context of type `R`, may signal errors of type `E`, and yields values of type `A`. However, the key distinction is that it can yield **zero or more values**.

Here are the possible scenarios for a `Stream`:

- **An Empty Stream**: It can end up empty, representing a stream with no values.
- **A Single-Element Stream**: It can represent a stream with just one value.
- **A Finite Stream of Elements**: It can represent a stream with a finite number of values.
- **An Infinite Stream of Elements**: It can represent a stream that continues indefinitely, essentially an infinite stream.

Let's see these scenarios in action:

```
import { Stream } from "effect"

// An empty Stream
const emptyStream = Stream.empty

// A Stream with a single number
const oneNumberValueStream = Stream.succeed(3)

// A Stream with a range of numbers from 1 to 10
const finiteNumberStream = Stream.range(1, 10)

// An infinite Stream of numbers starting from 1 and incrementing
const infiniteNumberStream = Stream.iterate(1, (n) => n + 1)
```

In summary, a `Stream` is a versatile tool for representing programs that may yield multiple values, making it suitable for a wide range of tasks, from processing finite lists to handling infinite sequences.

Scheduling Streams

Location: 400-guides/660-streaming/100-stream/700-scheduling

Introduce specific time intervals between stream element emissions with `Stream.schedule`. Learn to create structured pauses using scheduling combinators for precise control over stream timing.

schedule

When working with streams, you might need to introduce specific time intervals between each emission of stream elements. This can be achieved using the `Stream.schedule` combinator.

```
import { Stream, Schedule, Console, Effect } from "effect"

const stream = Stream.make(1, 2, 3, 4, 5).pipe(
  Stream.schedule(Schedule.spaced("1 second")),
  Stream.tap(Console.log)
)

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
1
2
3
4
5
{
  _id: "Chunk",
  values: [ 1, 2, 3, 4, 5 ]
}
*/
```

In this example, we've used the `Schedule.spaced("1 second")` schedule to introduce a one-second gap between each emission in the stream.

Resourceful Streams

Location: 400-guides/660-streaming/100-stream/300-resourceful-streams

Discover how to manage resources effectively in Effect's `Stream` module. Explore constructors tailored for lifting scoped resources, ensuring safe acquisition and release within streams. Dive into examples illustrating the use of `Stream.acquireRelease` for file operations, finalization for cleanup tasks, and `ensuring` for post-finalization actions. Master the art of resource management in streaming applications with Effect.

In the Stream module, you'll find that most of the constructors offer a special variant designed for lifting a scoped resource into a `Stream`. When you use these specific constructors, you're essentially creating streams that are inherently safe with regards to resource management. These constructors, before creating the stream, handle the resource acquisition, and after the stream's usage, they ensure its proper closure.

Stream also provides us with `Stream.acquireRelease` and `Stream.finalizer` constructors that share similarities with `Effect.acquireRelease` and `Effect.addFinalizer`. These tools empower us to perform cleanup or finalization tasks before the stream concludes its operation.

Acquire Release

In this section, we'll explore an example that demonstrates the use of `Stream.acquireRelease` when working with file operations.

```
import { Stream, Console, Effect } from "effect"

// Simulating File operations
const open = (filename: string) =>
  Effect.gen(function* () {
    yield* Console.log(`Opening ${filename}`)
    return {
      getLines: Effect.succeed(["Line 1", "Line 2", "Line 3"]),
      close: Console.log(`Closing ${filename}`)
    }
  })

const stream = Stream.acquireRelease(
  open("file.txt"),
  (file) => file.close
).pipe(Stream.flatMap((file) => file.getLines))

Effect.runPromise(Stream.runCollect(stream)).then(console.log)
/*
Output:
Opening file.txt
Closing file.txt
{
  _id: "Chunk",
  values: [
    [ "Line 1", "Line 2", "Line 3" ]
  ]
}
*/
```

In this code snippet, we're simulating file operations using the `open` function. The `Stream.acquireRelease` function is employed to ensure that the file is correctly opened and closed, and we then process the lines of the file using the acquired resource.

Finalization

In this section, we'll explore the concept of finalization in streams. Finalization allows us to execute a specific action before a stream ends. This can be particularly useful when we want to perform cleanup tasks or add final touches to a stream.

Imagine a scenario where our streaming application needs to clean up a temporary directory when it completes its execution. We can achieve this using the `Stream.finalizer` function:

```
import { Stream, Console, Effect } from "effect"

const application = Stream.fromEffect(Console.log("Application Logic."))

const deleteDir = (dir: string) => Console.log(`Deleting dir: ${dir}`)

const program = application.pipe(
  Stream.concat(
    Stream.finalizer(
      deleteDir("tmp").pipe(
        Effect.andThen(Console.log("Temporary directory was deleted."))
      )
    )
  )
)

Effect.runPromise(Stream.runCollect(program)).then(console.log)
/*
Output:
Application Logic.
Deleting dir: tmp
Temporary directory was deleted.
{
  _id: "Chunk",
  values: [ undefined, undefined ]
}
*/
```

In this code example, we start with our application logic represented by the `application` stream. We then use `Stream.finalizer` to define a finalization step, which deletes a temporary directory and logs a message. This ensures that the temporary directory is cleaned up properly when the application completes its execution.

Ensuring

In this section, we'll explore a scenario where we need to perform actions after the finalization of a stream. To achieve this, we can utilize the `Stream.ensuring` operator.

Consider a situation where our application has completed its primary logic and finalized some resources, but we also need to perform additional actions afterward. We can use `Stream.ensuring` for this purpose:

```
import { Stream, Console, Effect } from "effect"

const program = Stream.fromEffect(Console.log("Application Logic.")).pipe(
  Stream.concat(Stream.finalizer(Console.log("Finalizing the stream"))),
  Stream.ensuring(
    Console.log("Doing some other works after stream's finalization")
  )
)

Effect.runPromise(Stream.runCollect(program)).then(console.log)
/*
Output:
Application Logic.
Finalizing the stream
Doing some other works after stream's finalization
{
  _id: "Chunk",
  values: [ undefined, undefined ]
}
*/
```

In this code example, we start with our application logic represented by the `Application Logic.` message. We then use `Stream.finalizer` to specify the finalization step, which logs `Finalizing the stream`. After that, we use `Stream.ensuring` to indicate that we want to perform additional tasks after the stream's finalization, resulting in the message `Performing additional tasks after stream's finalization`. This ensures that our post-finalization actions are executed as expected.

SubscriptionRef

Location: 400-guides/660-streaming/300-subscriptionref

Explore the capabilities of `SubscriptionRef` in Effect, a specialized form of `SynchronizedRef`. Learn how it allows you to subscribe and receive updates on the current value and any changes made to that value. Understand the power of the `changes` stream, which facilitates observing the value and subsequent changes. Dive into practical examples demonstrating the use of `SubscriptionRef` in modeling shared state, especially in scenarios where multiple observers need to react to every change. Witness the seamless integration of `SubscriptionRef` with asynchronous tasks and discover how it enhances efficient state management in your programs.

A `SubscriptionRef<A>` is a specialized form of a SynchronizedRef. It allows us to subscribe and receive updates on the current value and any changes made to that value.

```
export interface SubscriptionRef<A> extends Synchronized.SynchronizedRef<A> {
  /**
   * A stream containing the current value of the `Ref` as well as all changes
   * to that value.
   */
  readonly changes: Stream<A>
}
```

You can perform all the usual operations on a `SubscriptionRef`, such as `get`, `set`, or `modify` to work with the current value.

The `changes` stream is where the magic happens. It lets you observe the current value and all subsequent changes. Each time you run this stream, you'll get the current value as of that moment and any changes that occurred afterward.

To create a `SubscriptionRef`, you can use the `make` constructor, specifying the initial value:

```
import { SubscriptionRef } from "effect"

const ref = SubscriptionRef.make(0)
```

A `SubscriptionRef` can be invaluable when modeling shared state, especially when multiple observers need to react to every change in that shared state. For example, in a functional reactive programming context, the `SubscriptionRef` value might represent a part of the application state, and each observer could update various user interface elements based on changes to that state.

To see how this works, let's create a simple example where a "server" repeatedly updates a value observed by multiple "clients":

```
import { Ref, Effect } from "effect"

const server = (ref: Ref.Ref<number>) =>
  Ref.update(ref, (n) => n + 1).pipe(Effect.forever)
```

Note that the `server` function operates on a regular `Ref` and doesn't need to know about `SubscriptionRef`. It simply updates a value.

```
import { Ref, Effect, Stream, Random } from "effect"

const server = (ref: Ref.Ref<number>) =>
  Ref.update(ref, (n) => n + 1).pipe(Effect.forever)
```

```
const client = (changes: Stream.Stream<number>) =>
  Effect.gen(function* () {
    const n = yield* Random.nextIntBetween(1, 10)
    const chunk = yield* Stream.runCollect(Stream.take(changes, n))
    return chunk
  })
```

Similarly, the `client` function only works with a `Stream` of values and doesn't concern itself with the source of these values.

To tie everything together, we start the server, launch multiple client instances in parallel, and then shut down the server when we're finished. We also create the `SubscriptionRef` in this process.

```
import { Ref, Effect, Stream, Random, SubscriptionRef, Fiber } from "effect"

const server = (ref: Ref.Ref<number>) =>
  Ref.update(ref, (n) => n + 1).pipe(Effect.forever)

const client = (changes: Stream.Stream<number>) =>
  Effect.gen(function* () {
    const n = yield* Random.nextIntBetween(1, 10)
    const chunk = yield* Stream.runCollect(Stream.take(changes, n))
    return chunk
  })

const program = Effect.gen(function* () {
  const ref = yield* SubscriptionRef.make(0)
  const serverFiber = yield* Effect.fork(server(ref))
  const clients = new Array(5).fill(null).map(() => client(ref.changes))
  const chunks = yield* Effect.all(clients, { concurrency: "unbounded" })
  yield* Fiber.interrupt(serverFiber)
  for (const chunk of chunks) {
    console.log(chunk)
  }
})

Effect.runPromise(program)
/*
Output:
{
  _id: "Chunk",
  values: [ 2, 3, 4 ]
}
{
  _id: "Chunk",
  values: [ 2 ]
}
{
  _id: "Chunk",
  values: [ 2, 3, 4, 5, 6, 7 ]
}
{
  _id: "Chunk",
  values: [ 2, 3, 4 ]
}
{
  _id: "Chunk",
  values: [ 2, 3, 4, 5, 6, 7, 8, 9 ]
}
*/
```

This setup ensures that each client observes the current value when it starts and receives all subsequent changes to the value.

Since the changes are represented as streams, you can easily build more complex programs using familiar stream operators. You can transform, filter, or merge these streams with other streams to achieve more sophisticated behavior.

Guides

Location: 400-guides/index

Guides

Semaphore

Location: 400-guides/640-concurrency/160-semaphore

Discover the power of semaphores in Effect, a synchronization mechanism that regulates access to shared resources and coordinates tasks in an asynchronous and concurrent environment. Delve into the fundamental concept of semaphores, learn how they function in Effect, and explore real-world examples showcasing their application in controlling asynchronous tasks. Gain insights into precise control over concurrency using permits and understand how semaphores elevate your ability to manage resources effectively.

A semaphore, in the context of programming, is a synchronization mechanism that allows you to control access to a shared resource. In Effect, semaphores are used to manage access to resources or coordinate tasks in an asynchronous and concurrent environment. Let's dive into the concept

of semaphores and how they work in Effect.

What is a Semaphore?

A semaphore is a generalization of a mutex. It has a certain number of **permits**, which can be held and released concurrently by different parties. Think of permits as tickets that allow entities (e.g., tasks or fibers) to access a shared resource or perform a specific operation. If there are no permits available and an entity tries to acquire one, it will be suspended until a permit becomes available.

Let's take a look at an example using asynchronous tasks:

```
import { Effect } from "effect"

const task = Effect.gen(function* () {
  yield* Effect.log("start")
  yield* Effect.sleep("2 seconds")
  yield* Effect.log("end")
})

const semTask = (sem: Effect.Semaphore) => sem.withPermits(1)(task)

const semTaskSeq = (sem: Effect.Semaphore) =>
  [1, 2, 3].map(() => semTask(sem).pipe(Effect.withLogSpan("elapsed")))

const program = Effect.gen(function* () {
  const mutex = yield* Effect.makeSemaphore(1)
  yield* Effect.all(semTaskSeq(mutex), { concurrency: "unbounded" })
})

Effect.runPromise(program)
/*
Output:
timestamp=... level=INFO fiber=#1 message=start elapsed=3ms
timestamp=... level=INFO fiber=#1 message=end elapsed=2010ms
timestamp=... level=INFO fiber=#2 message=start elapsed=2012ms
timestamp=... level=INFO fiber=#2 message=end elapsed=4017ms
timestamp=... level=INFO fiber=#3 message=start elapsed=4018ms
timestamp=... level=INFO fiber=#3 message=end elapsed=6026ms
*/
```

Here, we synchronize and control the execution of asynchronous tasks using a semaphore with one permit. When all permits are in use, additional tasks attempting to acquire permits will wait until some become available.

In another scenario, we create a semaphore with five permits. We then utilize `withPermits(n)` to acquire and release varying numbers of permits for each task:

```
import { Effect } from "effect"

const program = Effect.gen(function* () {
  const sem = yield* Effect.makeSemaphore(5)

  yield* Effect.forEach(
    [1, 2, 3, 4, 5],
    (n) =>
      sem
        .withPermits(n)(
          Effect.delay(Effect.log(`process: ${n}`), "2 seconds")
        )
        .pipe(Effect.withLogSpan("elasped")),
    { concurrency: "unbounded" }
  )
})

Effect.runPromise(program)
/*
Output:
timestamp=... level=INFO fiber=#1 message="process: 1" elasped=2011ms
timestamp=... level=INFO fiber=#2 message="process: 2" elasped=2017ms
timestamp=... level=INFO fiber=#3 message="process: 3" elasped=4020ms
timestamp=... level=INFO fiber=#4 message="process: 4" elasped=6025ms
timestamp=... level=INFO fiber=#5 message="process: 5" elasped=8034ms
*/
```

In this example, we show that you can acquire and release any number of permits with `withPermits(n)`. This flexibility allows for precise control over concurrency.

One crucial aspect to remember is that `withPermits` ensures that each acquisition is matched with an equivalent number of releases, regardless of whether the task succeeds, fails, or gets interrupted.

Concurrency Options

Location: 400-guides/640-concurrency/100-concurrency-options

Effect provides powerful options to manage the execution of effects, offering control over the concurrency of operations. Explore the `concurrency` option, a key factor in determining how many effects can run concurrently. This concise guide delves into sequential execution, numbered concurrency, unbounded concurrency, and the flexible inherit concurrency option. Learn to harness these options to optimize the performance of your Effect programs and tailor the concurrency behavior to your specific use cases.

Effect offers various options to manage how effects are executed and control the overall operation's result. These options help determine how many effects can run at the same time concurrently.

```
type Options = {
  readonly concurrency?: Concurrency
  /* ... other options ... */
}
```

In this section, we'll focus on the option that handles concurrency, which is the `concurrency` option with a type of `Concurrency`:

```
type Concurrency = number | "unbounded" | "inherit"
```

Let's understand the meaning of each configuration value.

<Info> The following examples use the `Effect.all` function, but the concept applies to many other Effect APIs that accept concurrency options, such as `Effect.forEach`. </Info>

Sequential Execution (Default)

By default, if you don't specify any concurrency option, effects will run sequentially, one after the other. This means each effect starts only after the previous one completes.

```
import { Effect, Duration } from "effect"

const makeTask = (n: number, delay: Duration.DurationInput) =>
  Effect.promise(
    () =>
      new Promise<void>((resolve) => {
        console.log(`start task${n}`)
        setTimeout(() => {
          console.log(`task${n} done`)
          resolve()
        }, Duration.toMillis(delay))
      })
  )

const task1 = makeTask(1, "200 millis")
const task2 = makeTask(2, "100 millis")

const sequential = Effect.all([task1, task2])

Effect.runPromise(sequential)
/*
Output:
start task1
task1 done
start task2 <-- task2 starts only after task1 completes
task2 done
*/
```

Numbered Concurrency

You can control the number of concurrent operations with the `concurrency` option. For example, with `concurrency: 2`, up to 2 effects will run simultaneously.

```
import { Effect, Duration } from "effect"

const makeTask = (n: number, delay: Duration.DurationInput) =>
  Effect.promise(
    () =>
      new Promise<void>((resolve) => {
        console.log(`start task${n}`)
        setTimeout(() => {
          console.log(`task${n} done`)
          resolve()
        }, Duration.toMillis(delay))
      })
  )

const task1 = makeTask(1, "200 millis")
const task2 = makeTask(2, "100 millis")
const task3 = makeTask(3, "210 millis")
const task4 = makeTask(4, "110 millis")
const task5 = makeTask(5, "150 millis")

const number = Effect.all([task1, task2, task3, task4, task5], {
  concurrency: 2
})

Effect.runPromise(number)
```

```
/*
Output:
start task1
start task2 <-- active tasks: task1, task2
task2 done
start task3 <-- active tasks: task1, task3
task1 done
start task4 <-- active tasks: task3, task4
task4 done
start task5 <-- active tasks: task3, task5
task3 done
task5 done
*/
```

Unbounded Concurrency

If you set `concurrency: "unbounded"`, as many effects as needed will run concurrently, without any specific limit.

```
import { Effect, Duration } from "effect"

const makeTask = (n: number, delay: Duration.DurationInput) =>
  Effect.promise(
    () =>
      new Promise<void>((resolve) => {
        console.log(`start task${n}`)
        setTimeout(() => {
          console.log(`task${n} done`)
          resolve()
        }, Duration.toMillis(delay))
      })
  )

const task1 = makeTask(1, "200 millis")
const task2 = makeTask(2, "100 millis")
const task3 = makeTask(3, "210 millis")
const task4 = makeTask(4, "110 millis")
const task5 = makeTask(5, "150 millis")

const unbounded = Effect.all([task1, task2, task3, task4, task5], {
  concurrency: "unbounded"
})

Effect.runPromise(unbounded)
/*
Output:
start task1
start task2
start task3
start task4
start task5
task2 done
task4 done
task5 done
task1 done
task3 done
*/
```

Inherit Concurrency

The `concurrency: "inherit"` option adapts based on context, controlled by `Effect.withConcurrency(number | "unbounded")`.

If there's no `Effect.withConcurrency` call, the default is `"unbounded"`. Otherwise, it inherits the configuration set by `Effect.withConcurrency`.

```
import { Effect, Duration } from "effect"

const makeTask = (n: number, delay: Duration.DurationInput) =>
  Effect.promise(
    () =>
      new Promise<void>((resolve) => {
        console.log(`start task${n}`)
        setTimeout(() => {
          console.log(`task${n} done`)
          resolve()
        }, Duration.toMillis(delay))
      })
  )

const task1 = makeTask(1, "200 millis")
const task2 = makeTask(2, "100 millis")
const task3 = makeTask(3, "210 millis")
const task4 = makeTask(4, "110 millis")
const task5 = makeTask(5, "150 millis")

const inherit = Effect.all([task1, task2, task3, task4, task5], {
  concurrency: "inherit"
})
```

```
Effect.runPromise(inherit)
/*
Output:
start task1
start task2
start task3
start task4
start task5
task2 done
task4 done
task5 done
task1 done
task3 done
*/
```

If you use `Effect.withConcurrency`, it will adopt that specific concurrency configuration.

```
import { Effect, Duration } from "effect"

const makeTask = (n: number, delay: Duration.DurationInput) =>
  Effect.promise(
    () =>
      new Promise<void>((resolve) => {
        console.log(`start task${n}`)
        setTimeout(() => {
          console.log(`task${n} done`)
          resolve()
        }, Duration.toMillis(delay))
      })
  )

const task1 = makeTask(1, "200 millis")
const task2 = makeTask(2, "100 millis")
const task3 = makeTask(3, "210 millis")
const task4 = makeTask(4, "110 millis")
const task5 = makeTask(5, "150 millis")

const inherit = Effect.all([task1, task2, task3, task4, task5], {
  concurrency: "inherit"
})

const withConcurrency = inherit.pipe(Effect.withConcurrency(2))

Effect.runPromise(withConcurrency)
/*
Output:
start task1
start task2 <-- active tasks: task1, task2
task2 done
start task3 <-- active tasks: task1, task3
task1 done
start task4 <-- active tasks: task3, task4
task4 done
start task5 <-- active tasks: task3, task5
task3 done
task5 done
*/
```

Introduction to Effect's Interruption Model

Location: 400-guides/640-concurrency/120-interruption-model

Explore the intricacies of Effect's interruption model, a crucial aspect in concurrent application development. Learn the nuances of handling fiber interruptions, including scenarios such as parent fibers terminating child fibers, racing fibers, user-initiated interruptions, and timeouts. Delve into the comparison between polling and asynchronous interruption, understanding the advantages of the latter in maintaining consistency and adhering to functional paradigms. Gain insights into when fibers get interrupted, providing examples and scenarios for a comprehensive understanding of this vital feature.

Handling Fiber Interruption

While developing concurrent applications, there are several cases that we need to *interrupt* the execution of other fibers, for example:

1. A parent fiber might start some child fibers to perform a task, and later the parent might decide that, it doesn't need the result of some or all of the child fibers.

2. Two or more fibers start race with each other. The fiber whose result is computed first wins, and all other fibers are no longer needed, and should be interrupted.

3. In interactive applications, a user may want to stop some already running tasks, such as clicking on the "stop" button to prevent downloading more files.

4. Computations that run longer than expected should be aborted by using timeout operations.

5. When we have an application that perform compute-intensive tasks based on the user inputs, if the user changes the input we should cancel the current task and perform another one.

Polling vs. Asynchronous Interruption

When it comes to interrupting fibers, a naive approach is to allow one fiber to forcefully terminate another fiber. However, this approach is not ideal because it can leave shared state in an inconsistent and unreliable state if the target fiber is in the middle of modifying that state. Therefore, it does not guarantee internal consistency of the shared mutable state.

Instead, there are two popular and valid solutions to tackle this problem:

1. **Semi-asynchronous Interruption (Polling for Interruption)**: Imperative languages often employ polling as a semi-asynchronous signaling mechanism, such as Java. In this model, a fiber sends an interruption request to another fiber. The target fiber continuously polls the interrupt status and checks whether it has received any interruption requests from other fibers. If an interruption request is detected, the target fiber terminates itself as soon as possible.

   With this solution, the fiber itself handles critical sections. So, if a fiber is in the middle of a critical section and receives an interruption request, it ignores the interruption and defers its handling until after the critical section.

   However, one drawback of this approach is that if the programmer forgets to poll regularly, the target fiber can become unresponsive, leading to deadlocks. Additionally, polling a global flag is not aligned with the functional paradigm followed by Effect.

2. **Asynchronous Interruption**: In asynchronous interruption, a fiber is allowed to terminate another fiber. The target fiber is not responsible for polling the interrupt status. Instead, during critical sections, the target fiber disables the interruptibility of those regions. This is a purely functional solution that doesn't require polling a global state. Effect adopts this solution for its interruption model, which is a fully asynchronous signaling mechanism.

   This mechanism overcomes the drawback of forgetting to poll regularly. It is also fully compatible with the functional paradigm because in a purely functional computation, we can abort the computation at any point, except during critical sections where interruption is disabled.

When Does a Fiber Get Interrupted?

There are several ways and situations in which fibers can be interrupted. Let's explore each one and provide examples to illustrate how to reproduce these scenarios.

Calling Effect.interrupt

A fiber can be interrupted by invoking the `Effect.interrupt` operator on that particular fiber.

Without interruptions

With interruptions

Interruption of Concurrent Effects

When we combine multiple concurrent effects using functions like `Effect.forEach`, it's important to note that if one of the effects is interrupted, all the other concurrent effects will also be interrupted. Let's take a look at an example:

In this example, we have an array `[1, 2, 3]` representing three concurrent tasks. We use `Effect.forEach` to iterate over each element and perform some operations. The `Effect.log` function is used to log messages indicating the start and completion of each task.

Looking at the output, we can see that the task with `n = 1` starts and completes successfully. However, the task with `n = 2` is interrupted using `Effect.interrupt` before it finishes. As a result, all the fibers are interrupted, and the program terminates with the message "All fibers interrupted without errors."

This example demonstrates how interruption works with concurrent effects. If one of the concurrent tasks is interrupted, it triggers the interruption of all the other concurrent tasks as well.

Concurrency

Location: 400-guides/640-concurrency/index

Concurrency

PubSub

Location: 400-guides/640-concurrency/150-pubsub

Dive into the world of `PubSub` with Effect, a powerful asynchronous message hub that facilitates seamless communication between publishers and subscribers. Learn the core operations, explore different types of pubsubs, and discover the optimal scenarios for their use. Understand the versatile operators available on pubsubs, from publishing multiple values to checking size and gracefully shutting down. Gain insights into the unique

qualities that set pubsubs apart and their equivalence to queues in various scenarios. Elevate your understanding of `PubSub` to enhance your asynchronous workflows.

In this guide, we'll explore the concept of a `PubSub`, which is an asynchronous message hub. It allows publishers to send messages to the pubsub, and subscribers can receive those messages.

Unlike a Queue, where each value offered to the queue can be taken by **one** taker, each value published to a pubsub can be received by **all** subscribers.

Whereas a Queue represents the optimal solution to the problem of how to **distribute** values, a `PubSub` represents the optimal solution to the problem of how to **broadcast** them.

Basic Operations

The core operations of a `PubSub` are `PubSub.publish` and `PubSub.subscribe`:

- The `publish` operation sends a message of type `A` to the pubsub. It returns an effect that indicates whether the message was successfully published.
- The `subscribe` operation returns a scoped effect that allows you to subscribe to the pubsub. It automatically unsubscribes when the scope is closed. Within the scope, you gain access to a `Dequeue`, which is essentially a `Queue` for dequeuing messages published to the pubsub.

Let's look at an example to understand how to use a pubsub:

```
import { Effect, PubSub, Queue, Console } from "effect"

const program = PubSub.bounded<string>(2).pipe(
  Effect.andThen((pubsub) =>
    Effect.scoped(
      Effect.gen(function* () {
        const dequeue1 = yield* PubSub.subscribe(pubsub)
        const dequeue2 = yield* PubSub.subscribe(pubsub)
        yield* PubSub.publish(pubsub, "Hello from a PubSub!")
        yield* Queue.take(dequeue1).pipe(Effect.andThen(Console.log))
        yield* Queue.take(dequeue2).pipe(Effect.andThen(Console.log))
      })
    )
  )
)

Effect.runPromise(program)
/*
Output:
Hello from a PubSub!
Hello from a PubSub!
*/
```

It's important to note that a subscriber will only receive messages published to the pubsub while it's subscribed. To ensure that a specific message reaches a subscriber, make sure that the subscription has been established before publishing the message.

Creating PubSubs

You can create a pubsub using various constructors provided by the PubSub module:

Bounded PubSub

A bounded pubsub applies back pressure to publishers when it's at capacity, meaning publishers will block if the pubsub is full.

```
import { PubSub } from "effect"

const boundedPubSub = PubSub.bounded<string>(2)
```

Back pressure ensures that all subscribers receive all messages while they are subscribed. However, it can lead to slower message delivery if a subscriber is slow.

Dropping PubSub

A dropping pubsub simply discards values if it's full. The `PubSub.publish` function will return `false` when the pubsub is full.

```
import { PubSub } from "effect"

const droppingPubSub = PubSub.dropping<string>(2)
```

In a dropping pubsub, publishers can continue to publish new values, but subscribers are not guaranteed to receive all messages.

Sliding PubSub

A sliding pubsub drops the oldest value when it's full, ensuring that publishing always succeeds immediately.

```
import { PubSub } from "effect"
```

```
const slidingPubSub = PubSub.sliding<string>(2)
```

A sliding pubsub prevents slow subscribers from impacting the message delivery rate. However, there's still a risk that slow subscribers may miss some messages.

Unbounded PubSub

An unbounded pubsub can never be full, and publishing always succeeds immediately.

```
import { PubSub } from "effect"

const unboundedPubSub = PubSub.unbounded<string>()
```

Unbounded pubsubs guarantee that all subscribers receive all messages without slowing down message delivery. However, they can grow indefinitely if messages are published faster than they are consumed.

Generally, it's recommended to use bounded, dropping, or sliding pubsubs unless you have specific use cases for unbounded pubsubs.

Operators On PubSubs

PubSubs support various operations similar to those available on queues.

Publishing Multiple Values

You can use the `PubSub.publishAll` operator to publish multiple values to the pubsub at once:

```
import { Effect, PubSub, Queue, Console } from "effect"

const program = PubSub.bounded<string>(2).pipe(
  Effect.andThen((pubsub) =>
    Effect.scoped(
      Effect.gen(function* () {
        const dequeue = yield* PubSub.subscribe(pubsub)
        yield* PubSub.publishAll(pubsub, ["Message 1", "Message 2"])
        yield* Queue.takeAll(dequeue).pipe(Effect.andThen(Console.log))
      })
    )
  )
)

Effect.runPromise(program)
/*
Output:
{
  _id: "Chunk",
  values: [ "Message 1", "Message 2" ]
}
*/
```

Checking Size

You can determine the capacity and current size of the pubsub using `PubSub.capacity` and `PubSub.size`:

```
import { Effect, PubSub, Console } from "effect"

const program = PubSub.bounded<number>(2).pipe(
  Effect.tap((pubsub) => Console.log(`capacity: ${PubSub.capacity(pubsub)}`)),
  Effect.tap((pubsub) =>
    PubSub.size(pubsub).pipe(
      Effect.andThen((size) => Console.log(`size: ${size}`))
    )
  )
)

Effect.runPromise(program)
/*
Output:
capacity: 2
size: 0
*/
```

Note that `capacity` returns a `number` because the capacity is set at pubsub creation and never changes. In contrast, `size` returns an effect that determines the current size of the pubsub since the number of messages in the pubsub can change over time.

Shutting Down a PubSub

You can shut down a pubsub using `PubSub.shutdown`, check if it's shut down with `PubSub.isShutdown`, or await its shutdown with `PubSub.awaitShutdown`. Shutting down a pubsub also shuts down all associated queues, ensuring the proper propagation of the shutdown signal.

PubSub as an Enqueue

As you can see, the operators on `PubSub` are identical to the ones on Queue with the exception of `PubSub.publish` and `PubSub.subscribe` replacing `Queue.offer` and `Queue.take`. So if you know how to use a `Queue`, you already know how to use a `PubSub`.

In fact, a `PubSub` can be viewed as a `Queue` that can only be written to:

```
interface PubSub<A> extends Queue.Enqueue<A> {}
```

Here, the `Enqueue` type represents a queue that can only be enqueued. Enqueuing to the queue publishes a value to the pubsub, and actions like shutting down the queue also shut down the pubsub.

This versatility allows you to use a `PubSub` wherever you currently use a `Queue` that you only write to.

Queue

Location: 400-guides/640-concurrency/140-queue

Explore the lightweight, in-memory `Queue` in Effect, designed for composable and transparent back-pressure. Learn about its fully asynchronous, purely-functional, and type-safe characteristics. Delve into basic operations, creating different types of queues, and efficiently adding and consuming items. Discover how to shut down a queue gracefully and handle exclusive capabilities with offer-only and take-only queues. Enhance your understanding of `Queue` and leverage its power for seamless coordination in your asynchronous workflows.

A `Queue` is a lightweight in-memory queue built on Effect with composable and transparent back-pressure. It is fully asynchronous (no locks or blocking), purely-functional and type-safe.

Basic Operations

A `Queue<A>` stores values of type `A` and provides two fundamental operations:

- `Queue.offer`: This operation adds a value of type `A` to the `Queue`.
- `Queue.take`: It removes and returns the oldest value from the `Queue`.

Here's an example demonstrating these basic operations:

```
import { Effect, Queue } from "effect"

const program = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(100)
  yield* Queue.offer(queue, 1) // Add 1 to the queue
  const value = yield* Queue.take(queue) // Retrieve and remove the oldest value
  return value
})

Effect.runPromise(program).then(console.log) // Output: 1
```

Creating a Queue

A `Queue` can have bounded (limited capacity) or unbounded storage. Depending on your requirements, you can choose from various strategies to handle new values when the queue reaches its capacity.

Bounded Queue

A bounded queue provides back-pressure when it's full. This means that if the queue is full, any attempt to add more items will be suspended until there's space available.

```
import { Queue } from "effect"

// Creating a bounded queue with a capacity of 100
const boundedQueue = Queue.bounded<number>(100)
```

Dropping Queue

A dropping queue simply drops new items when it's full. It doesn't wait for space to become available.

```
import { Queue } from "effect"

// Creating a dropping queue with a capacity of 100
const droppingQueue = Queue.dropping<number>(100)
```

Sliding Queue

A sliding queue removes old items when it's full to accommodate new ones.

```
import { Queue } from "effect"

// Creating a sliding queue with a capacity of 100
const slidingQueue = Queue.sliding<number>(100)
```

Unbounded Queue

An unbounded queue has no capacity limit.

```
import { Queue } from "effect"

// Creating an unbounded queue
const unboundedQueue = Queue.unbounded<number>()
```

Adding Items to a Queue

To add a value to the queue, you can use the `Queue.offer` operation:

```
import { Effect, Queue } from "effect"

const program = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(100)
  yield* Queue.offer(queue, 1) // put 1 in the queue
})
```

If you're using a back-pressured queue and it's full, the `offer` operation might suspend. In such cases, you can use `Effect.fork` to wait in a different execution context (fiber).

```
import { Effect, Queue, Fiber } from "effect"

const program = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(1)
  yield* Queue.offer(queue, 1)
  const fiber = yield* Effect.fork(Queue.offer(queue, 2)) // will be suspended because the queue is full
  yield* Queue.take(queue)
  yield* Fiber.join(fiber)
})
```

You can also add multiple values at once using `Queue.offerAll`:

```
import { Effect, Queue, Array } from "effect"

const program = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(100)
  const items = Array.range(1, 10)
  yield* Queue.offerAll(queue, items)
  return yield* Queue.size(queue)
})

Effect.runPromise(program).then(console.log) // Output: 10
```

Consuming Items from a Queue

The `Queue.take` operation removes the oldest item from the queue and returns it. If the queue is empty, it will suspend and resume only when an item is added to the queue. You can also use `Effect.fork` to wait for the value in a different execution context (fiber).

```
import { Effect, Queue, Fiber } from "effect"

const oldestItem = Effect.gen(function* () {
  const queue = yield* Queue.bounded<string>(100)
  const fiber = yield* Effect.fork(Queue.take(queue)) // will be suspended because the queue is empty
  yield* Queue.offer(queue, "something")
  const value = yield* Fiber.join(fiber)
  return value
})

Effect.runPromise(oldestItem).then(console.log) // Output: something
```

You can retrieve the first item using `Queue.poll`. If the queue is empty, you'll get `None`; otherwise, the top item will be wrapped in `Some`.

```
import { Effect, Queue } from "effect"

const polled = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(100)
  yield* Queue.offer(queue, 10)
  yield* Queue.offer(queue, 20)
  const head = yield* Queue.poll(queue)
  return head
})

Effect.runPromise(polled).then(console.log)
/*
Output:
{
  _id: "Option",
  _tag: "Some",
  value: 10
}
*/
```

You can retrieve multiple items at once using `Queue.takeUpTo`. If the queue doesn't have enough items to return, it will return all the available items without waiting for more offers.

```
import { Effect, Queue } from "effect"

const polled = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(100)
  yield* Queue.offer(queue, 10)
  yield* Queue.offer(queue, 20)
  yield* Queue.offer(queue, 30)
  const chunk = yield* Queue.takeUpTo(queue, 2)
  return chunk
})

Effect.runPromise(polled).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 10, 20 ]
}
*/
```

Similarly, you can retrieve all items at once using `Queue.takeAll`. It returns immediately, providing an empty collection if the queue is empty.

```
import { Effect, Queue } from "effect"

const polled = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(100)
  yield* Queue.offer(queue, 10)
  yield* Queue.offer(queue, 20)
  yield* Queue.offer(queue, 30)
  const chunk = yield* Queue.takeAll(queue)
  return chunk
})

Effect.runPromise(polled).then(console.log)
/*
Output:
{
  _id: "Chunk",
  values: [ 10, 20, 30 ]
}
*/
```

Shutting Down a Queue

With `Queue.shutdown`, you can interrupt all fibers that are suspended on `offer*` or `take*`. It also empties the queue and causes all future `offer*` and `take*` calls to terminate immediately.

```
import { Effect, Queue, Fiber } from "effect"

const program = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(3)
  const fiber = yield* Effect.fork(Queue.take(queue))
  yield* Queue.shutdown(queue) // will interrupt fiber
  yield* Fiber.join(fiber) // will terminate
})
```

You can use `Queue.awaitShutdown` to execute an effect when the queue is shut down. This function waits until the queue is shut down, and if it's already shut down, it resumes immediately.

```
import { Effect, Queue, Fiber, Console } from "effect"

const program = Effect.gen(function* () {
  const queue = yield* Queue.bounded<number>(3)
  const fiber = yield* Effect.fork(
    Queue.awaitShutdown(queue).pipe(
      Effect.andThen(Console.log("shutting down"))
    )
  )
  yield* Queue.shutdown(queue)
  yield* Fiber.join(fiber)
})

Effect.runPromise(program) // Output: shutting down
```

Offer-only / Take-only Queues

In some situations, you may need specific parts of your code to have exclusive capabilities, such as only offering values (`Enqueue`) or only taking values (`Dequeue`) from a queue. Effect provides a straightforward way to achieve this.

All operations related to offering values are defined by the `Enqueue` interface. Here's an example of how to use it:

```
import { Queue } from "effect"

const send = (offerOnlyQueue: Queue.Enqueue<number>, value: number) => {
  // This enqueue can only be used to offer values

  // @ts-expect-error
  Queue.take(offerOnlyQueue)

  // Ok
  return Queue.offer(offerOnlyQueue, value)
}
```

Similarly, all operations related to taking values are defined by the `Dequeue` interface. Here's an example:

```
import { Queue } from "effect"

const receive = (takeOnlyQueue: Queue.Dequeue<number>) => {
  // This dequeue can only be used to take values

  // @ts-expect-error
  Queue.offer(takeOnlyQueue, 1)

  // Ok
  return Queue.take(takeOnlyQueue)
}
```

The `Queue` type extends both `Enqueue` and `Dequeue`, allowing you to conveniently pass it to different parts of your code where you want to enforce either `Enqueue` or `Dequeue` behavior:

```
import { Effect, Queue } from "effect"

const send = (offerOnlyQueue: Queue.Enqueue<number>, value: number) => {
  return Queue.offer(offerOnlyQueue, value)
}

const receive = (takeOnlyQueue: Queue.Dequeue<number>) => {
  return Queue.take(takeOnlyQueue)
}

const program = Effect.gen(function* () {
  const queue = yield* Queue.unbounded<number>()

  // Offer values to the queue
  yield* send(queue, 1)
  yield* send(queue, 2)

  // Take values from the queue
  console.log(yield* receive(queue))
  console.log(yield* receive(queue))
})

Effect.runPromise(program)
/*
Output:
1
2
*/
```

Basic Concurrency

Location: 400-guides/640-concurrency/50-basic-concurrency

Basic Concurrency

Effect is a highly concurrent framework powered by fibers. Fibers are lightweight **virtual threads** with resource-safe cancellation capabilities, enabling many features in Effect.

In this section, you will learn the basics of fibers and get familiar with some of the powerful high-level operators that utilize fibers.

What Are Virtual Threads?

JavaScript is inherently single-threaded, meaning it executes code in a single sequence of instructions. However, modern JavaScript environments use an event loop to manage asynchronous operations, creating the illusion of multitasking. In this context, virtual threads, or fibers, are logical threads simulated by the Effect runtime. They allow concurrent execution without relying on true multi-threading, which is not natively supported in JavaScript.

Fibers

All effects in Effect are executed by fibers. If you didn't create the fiber yourself, it was created by an operation you're using (if it's concurrent) or by the Effect runtime system.

Even if you write "single-threaded" code with no concurrent operations, there will always be at least one fiber: the "main" fiber that executes your effect.

Effect fibers have a well-defined lifecycle based on the effect they are executing.

Every fiber exits with either a failure or success, depending on whether the effect it is executing fails or succeeds.

Effect fibers have unique identities, local state, and a status (such as done, running, or suspended).

The Fiber Data Type

The Fiber data type in Effect represents a "handle" on the execution of an effect.

The `Fiber<A, E>` data type has two type parameters:

- **A (Success Type)**: The type of value the fiber may succeed with.
- **E (Failure Type)**: The type of value the fiber may fail with.

Fibers do not have an `R` type parameter because they only execute effects that have already had their requirements provided to them.

Forking Effects

One of the fundamental ways to create a fiber is by forking an existing effect. When you fork an effect, it starts executing the effect on a new fiber, giving you a reference to this newly-created fiber.

The following code demonstrates how to create a single fiber using the `fork` function. This fiber will execute the function `fib(100)` independently of the main fiber:

```
import { Effect } from "effect"

const fib = (n: number): Effect.Effect<number> =>
  Effect.suspend(() => {
    if (n <= 1) {
      return Effect.succeed(n)
    }
    return fib(n - 1).pipe(Effect.zipWith(fib(n - 2), (a, b) => a + b))
  })

const fib10Fiber = Effect.fork(fib(10))
```

Joining Fibers

A common operation with fibers is joining them using the `Fiber.join` function. This function returns an `Effect` that will succeed or fail based on the outcome of the fiber it joins:

```
import { Effect, Fiber } from "effect"

const fib = (n: number): Effect.Effect<number> =>
  Effect.suspend(() => {
    if (n <= 1) {
      return Effect.succeed(n)
    }
    return fib(n - 1).pipe(Effect.zipWith(fib(n - 2), (a, b) => a + b))
  })

const fib10Fiber = Effect.fork(fib(10))

const program = Effect.gen(function* () {
  const fiber = yield* fib10Fiber
  const n = yield* Fiber.join(fiber)
  console.log(n)
})

Effect.runPromise(program) // 55
```

Awaiting Fibers

Another useful function for fibers is `Fiber.await`. This function returns an effect containing an Exit value, which provides detailed information about how the fiber completed.

```
import { Effect, Fiber } from "effect"

const fib = (n: number): Effect.Effect<number> =>
  Effect.suspend(() => {
    if (n <= 1) {
      return Effect.succeed(n)
    }
    return fib(n - 1).pipe(Effect.zipWith(fib(n - 2), (a, b) => a + b))
  })

const fib10Fiber = Effect.fork(fib(10))

const program = Effect.gen(function* () {
```

```
  const fiber = yield* fib10Fiber
  const exit = yield* Fiber.await(fiber)
  console.log(exit)
})

Effect.runPromise(program) // { _id: 'Exit', _tag: 'Success', value: 55 }
```

Interrupting Fibers

If a fiber's result is no longer needed, it can be interrupted, which immediately terminates the fiber and safely releases all resources by running all finalizers.

Similar to `Fiber.await`, `Fiber.interrupt` returns an Exit` value describing how the fiber completed.

```
import { Effect, Fiber } from "effect"

const program = Effect.gen(function* () {
  const fiber = yield* Effect.fork(Effect.forever(Effect.succeed("Hi!")))
  const exit = yield* Fiber.interrupt(fiber)
  console.log(exit)
})

Effect.runPromise(program)
/*
Output
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: {
    _id: 'Cause',
    _tag: 'Interrupt',
    fiberId: {
      _id: 'FiberId',
      _tag: 'Runtime',
      id: 0,
      startTimeMillis: 1715787137490
    }
  }
}
*/
```

By design, the effect returned by `Fiber.interrupt` does not resume until the fiber has completed, ensuring that your code does not start new fibers until the old one has terminated. This behavior, often called "back-pressuring," can be overridden if needed.

If you do not need back-pressuring, you can fork the interruption itself into a new fiber:

```
import { Effect, Fiber } from "effect"

const program = Effect.gen(function* () {
  const fiber = yield* Effect.fork(Effect.forever(Effect.succeed("Hi!")))
  const _ = yield* Effect.fork(Fiber.interrupt(fiber))
})
```

There is also a shorthand for background interruption called `Fiber.interruptFork`.

```
import { Effect, Fiber } from "effect"

const program = Effect.gen(function* () {
  const fiber = yield* Effect.fork(Effect.forever(Effect.succeed("Hi!")))
  const _ = yield* Fiber.interruptFork(fiber)
})
```

Note: It is also possible to perform interruptions using the high-level API `Effect.interrupt`. For more information, see Effect.interrupt.

Composing Fibers

The `Fiber.zip` and `Fiber.zipWith` functions allow you to combine two fibers into a single fiber. The resulting fiber produces the results of both input fibers. If either of the input fibers fails, the composed fiber will also fail.

Here's an example using `Fiber.zip`:

```
import { Effect, Fiber } from "effect"

const program = Effect.gen(function* () {
  const fiber1 = yield* Effect.fork(Effect.succeed("Hi!"))
  const fiber2 = yield* Effect.fork(Effect.succeed("Bye!"))
  const fiber = Fiber.zip(fiber1, fiber2)
  const tuple = yield* Fiber.join(fiber)
  console.log(tuple)
})

Effect.runPromise(program)
/*
Output:
[ 'Hi!', 'Bye!' ]
*/
```

Another way to compose fibers is with the `Fiber.orElse` function. This function allows you to specify an alternative fiber that will be executed if the first fiber fails. If the first fiber succeeds, the composed fiber will return its result. If the first fiber fails, the composed fiber will complete with the result of the second fiber, regardless of whether it succeeds or fails.

Here's an example using `Fiber.orElse`:

```
import { Effect, Fiber } from "effect"

const program = Effect.gen(function* () {
  const fiber1 = yield* Effect.fork(Effect.fail("Uh oh!"))
  const fiber2 = yield* Effect.fork(Effect.succeed("Hurray!"))
  const fiber = Fiber.orElse(fiber1, fiber2)
  const message = yield* Fiber.join(fiber)
  console.log(message)
})

Effect.runPromise(program)
/*
Output:
Hurray!
*/
```

Concurrency Options

Effect provides many functions that accept Concurrency Options to help you identify opportunities to parallelize your code.

For example, the standard `Effect.zip` function combines two effects sequentially. However, there is also a concurrent version, `Effect.zip({_, _, { concurrent: true })`, which combines two effects concurrently.

In the following example, we use `Effect.zip` to run two tasks sequentially. The first task takes 1 second, and the second task takes 2 seconds, resulting in a total duration of approximately 3 seconds:

```
import { Effect, Console } from "effect"

const task1 = Effect.delay(Console.log("task1"), "1 second")
const task2 = Effect.delay(Console.log("task2"), "2 seconds")

const program = Effect.zip(task1, task2)

Effect.runPromise(Effect.timed(program)).then(([duration]) =>
  console.log(String(duration))
)
/*
Output:
task1
task2
Duration(3s 5ms 369875ns)
*/
```

In this example, we use the concurrent version of `Effect.zip` to run two tasks concurrently. The total duration will be approximately equal to the duration of the longest task, which is 2 seconds:

```
import { Effect, Console } from "effect"

const task1 = Effect.delay(Console.log("task1"), "1 second")
const task2 = Effect.delay(Console.log("task2"), "2 seconds")

const program = Effect.zip(task1, task2, { concurrent: true })

Effect.runPromise(Effect.timed(program)).then(([duration]) =>
  console.log(String(duration))
)
/*
Output:
task1
task2
Duration(2s 8ms 179666ns)
*/
```

Racing

The `Effect.race` function lets you race multiple effects concurrently and returns the result of the first one that successfully completes. Here's an example:

```
import { Effect } from "effect"

const task1 = Effect.delay(Effect.fail("task1"), "1 second")
const task2 = Effect.delay(Effect.succeed("task2"), "2 seconds")

const program = Effect.race(task1, task2)

Effect.runPromise(program).then(console.log)
/*
Output:
```

```
task2
*/
```

In this example, `task1` is set to fail after 1 second, while `task2` is set to succeed after 2 seconds. The `Effect.race` function runs both tasks concurrently, and since `task2` is the first to succeed, its result is returned.

If you need to handle the first effect to complete, whether it succeeds or fails, you can use the `Effect.either` function. This function wraps the result in an Either type, allowing you to see if the outcome was a success (`Right`) or a failure (`Left`):

```
import { Effect } from "effect"

const task1 = Effect.delay(Effect.fail("task1"), "1 second")
const task2 = Effect.delay(Effect.succeed("task2"), "2 seconds")

const program = Effect.race(Effect.either(task1), Effect.either(task2))

Effect.runPromise(program).then(console.log)
/*
Output:
{ _id: 'Either', _tag: 'Left', left: 'task1' }
*/
```

In this example, `task1` fails after 1 second, and `task2` succeeds after 2 seconds. By using `Effect.either`, the program returns the result of `task1`, showing that it was a failure (`Left`).

Timeout

When working with asynchronous tasks, it's often important to ensure that they complete within a reasonable time. Effect provides a convenient way to enforce time limits on effects using the `Effect.timeout` function. This function returns a new effect that will fail with a `TimeoutException` if the original effect does not complete within the specified duration.

Here's an example demonstrating how to use `Effect.timeout`:

```
import { Effect } from "effect"

const task = Effect.delay(Effect.succeed("task1"), "10 seconds")

const program = Effect.timeout(task, "2 seconds")

Effect.runPromise(program)
/*
throws:
TimeoutException
*/
```

In this example, `task` is an effect that succeeds after 10 seconds. By wrapping `task` with `Effect.timeout` and specifying a timeout of 2 seconds, the resulting program will fail with a `TimeoutException` because the task takes longer than the allowed time.

If an effect times out, the `effect` library automatically interrupts it to prevent it from continuing to execute in the background. This interruption ensures efficient use of resources by stopping unnecessary work.

Deferred

Location: 400-guides/640-concurrency/130-deferred

Explore the power of `Deferred` in Effect, a specialized subtype of `Effect` that acts as a one-time variable with unique characteristics. Discover how `Deferred` serves as a synchronization tool for managing asynchronous operations, allowing coordination between different parts of your code. Learn common use cases, including coordinating fibers, synchronization, handing over work, and suspending execution. Dive into operations such as creating, awaiting, completing, and polling `Deferred`, providing practical examples and scenarios to enhance your understanding of this powerful tool.

A `Deferred<A, E>` is a special subtype of `Effect` that acts as a variable, but with some unique characteristics. It can only be set once, making it a powerful synchronization tool for managing asynchronous operations.

A `Deferred` is essentially a synchronization primitive that represents a value that may not be available immediately. When you create a `Deferred`, it starts with an empty value. Later on, you can complete it exactly once with either a success value (`A`) or a failure value (`E`). Once completed, a `Deferred` can never be modified or emptied again.

Common Use Cases

`Deferred` becomes incredibly useful when you need to wait for something specific to happen in your program. It's ideal for scenarios where you want one part of your code to signal another part when it's ready. Here are a few common use cases:

- **Coordinating Fibers**: When you have multiple concurrent tasks (fibers) and need to coordinate their actions, `Deferred` can help one fiber signal to another when it has completed its task.
- **Synchronization**: Anytime you want to ensure that one piece of code doesn't proceed until another piece of code has finished its work, `Deferred` can provide the synchronization you need.

- **Handing Over Work**: You can use `Deferred` to hand over work from one fiber to another. For example, one fiber can prepare some data, and then a second fiber can continue processing it.

- **Suspending Execution**: When you want a fiber to pause its execution until some condition is met, a `Deferred` can be used to block it until the condition is satisfied.

When a fiber calls `await` on a `Deferred`, it essentially blocks until that `Deferred` is completed with either a value or an error. Importantly, in Effect, blocking fibers don't actually block the main thread; they block only semantically. While one fiber is blocked, the underlying thread can execute other fibers, ensuring efficient concurrency.

A `Deferred` in Effect is conceptually similar to JavaScript's `Promise`. The key difference is that `Deferred` has two type parameters (`E` and `A`) instead of just one. This allows `Deferred` to represent both successful results (`A`) and errors (`E`).

Operations

Creating

You can create a `Deferred` using `Deferred.make<A, E>()`. This returns an `Effect<Deferred<A, E>>`, which describes the creation of a `Deferred`. Note that `Deferred`s can only be created within an `Effect` because creating them involves effectful memory allocation, which must be managed safely within an `Effect`.

Awaiting

To retrieve a value from a `Deferred`, you can use `Deferred.await`. This operation suspends the calling fiber until the `Deferred` is completed with a value or an error.

```
import { Effect, Deferred } from "effect"

const effectDeferred = Deferred.make<string, Error>()

const effectGet = effectDeferred.pipe(
  Effect.andThen((deferred) => Deferred.await(deferred))
)
```

Completing

You can complete a `Deferred<A, E>` in different ways:

There are several ways to complete a `Deferred`:

- `Deferred.succeed`: Completes the `Deferred` successfully with a value of type `A`.
- `Deferred.done`: Completes the `Deferred` with an `Exit<A, E>` type.
- `Deferred.complete`: Completes the `Deferred` with the result of an effect `Effect<A, E>`.
- `Deferred.completeWith`: Completes the `Deferred` with an effect `Effect<A, E>`. This effect will be executed by each waiting fiber, so use it carefully.
- `Deferred.fail`: Fails the `Deferred` with an error of type `E`.
- `Deferred.die`: Defects the `Deferred` with a user-defined error.
- `Deferred.failCause`: Fails or defects the `Deferred` with a `Cause<E>`.
- `Deferred.interrupt`: Interrupts the `Deferred`. This can be used to forcefully stop or interrupt the waiting fibers.

Here's an example that demonstrates the usage of these completion methods:

```
import { Effect, Deferred, Exit, Cause } from "effect"

const program = Effect.gen(function* () {
  const deferred = yield* Deferred.make<number, string>()

  // Completing the Deferred in various ways
  yield* Deferred.succeed(deferred, 1).pipe(Effect.fork)
  yield* Deferred.complete(deferred, Effect.succeed(2)).pipe(Effect.fork)
  yield* Deferred.completeWith(deferred, Effect.succeed(3)).pipe(Effect.fork)
  yield* Deferred.done(deferred, Exit.succeed(4)).pipe(Effect.fork)
  yield* Deferred.fail(deferred, "5").pipe(Effect.fork)
  yield* Deferred.failCause(deferred, Cause.die(new Error("6"))).pipe(
    Effect.fork
  )
  yield* Deferred.die(deferred, new Error("7")).pipe(Effect.fork)
  yield* Deferred.interrupt(deferred).pipe(Effect.fork)

  // Awaiting the Deferred to get its value
  const value = yield* Deferred.await(deferred)
  return value
})

Effect.runPromise(program).then(console.log, console.error) // Output: 1
```

When you complete a `Deferred`, it results in an `Effect<boolean>`. This effect returns `true` if the `Deferred` value has been set and `false` if it was already set before completion. This can be useful for checking the state of the `Deferred`.

Here's an example demonstrating the state change of a `Deferred`:

```
import { Effect, Deferred } from "effect"

const program = Effect.gen(function* () {
  const deferred = yield* Deferred.make<number, string>()
  const b1 = yield* Deferred.fail(deferred, "oh no!")
  const b2 = yield* Deferred.succeed(deferred, 1)
  return [b1, b2]
})

Effect.runPromise(program).then(console.log) // Output: [ true, false ]
```

Polling

Sometimes, you may want to check whether a `Deferred` has been completed without causing the fiber to suspend. To achieve this, you can use the `Deferred.poll` method. Here's how it works:

- `Deferred.poll` returns an `Option<Effect<A, E>>`.
  - If the `Deferred` is not yet completed, it returns `None`.
  - If the `Deferred` is completed, it returns `Some`, which contains the result or error.

Additionally, you can use the `Deferred.isDone` method, which returns an `Effect<boolean>`. This effect evaluates to `true` if the `Deferred` is already completed, allowing you to quickly check the completion status.

Here's a practical example:

```
import { Effect, Deferred } from "effect"

const program = Effect.gen(function* () {
  const deferred = yield* Deferred.make<number, string>()

  // Polling the Deferred
  const done1 = yield* Deferred.poll(deferred)

  // Checking if the Deferred is already completed
  const done2 = yield* Deferred.isDone(deferred)

  return [done1, done2]
})

Effect.runPromise(program).then(console.log) // Output: [ none(), false ]
```

In this example, we first create a `Deferred` and then use `Deferred.poll` to check its completion status. Since it's not completed yet, `done1` is `none()`. We also use `Deferred.isDone` to confirm that the `Deferred` is indeed not completed, so `done2` is `false`.

Example: Using Deferred to Coordinate Two Fibers

Here's a scenario where we use a `Deferred` to hand over a value between two fibers:

```
import { Effect, Deferred, Fiber } from "effect"

const program = Effect.gen(function* () {
  const deferred = yield* Deferred.make<string, string>()

  // Fiber A: Set the Deferred value after waiting for 1 second
  const sendHelloWorld = Effect.gen(function* () {
    yield* Effect.sleep("1 second")
    return yield* Deferred.succeed(deferred, "hello world")
  })

  // Fiber B: Wait for the Deferred and print the value
  const getAndPrint = Effect.gen(function* () {
    const s = yield* Deferred.await(deferred)
    console.log(s)
    return s
  })

  // Run both fibers concurrently
  const fiberA = yield* Effect.fork(sendHelloWorld)
  const fiberB = yield* Effect.fork(getAndPrint)

  // Wait for both fibers to complete
  return yield* Fiber.join(Fiber.zip(fiberA, fiberB))
})

Effect.runPromise(program).then(console.log, console.error)
/*
Output:
hello world
[ true, "hello world" ]
*/
```

In this example, we have two fibers, `fiberA` and `fiberB`, that communicate using a `Deferred`:

- `fiberA` sets the `Deferred` value to "hello world" after waiting for 1 second.
- `fiberB` waits for the `Deferred` to be completed and then prints the received value to the console.

By running both fibers concurrently and using the `Deferred` as a synchronization point, we can ensure that `fiberB` only proceeds after `fiberA` has completed its task. This coordination mechanism allows you to hand over values or coordinate work between different parts of your program effectively.

Fibers

Location: 400-guides/640-concurrency/110-fibers

Discover the power of fibers in Effect, providing a lightweight and efficient way to manage concurrency and asynchronous tasks. Learn the fundamentals of fibers, their role in multitasking, and how they contribute to responsive applications. Explore the creation of fibers, their lifetimes, and various forking strategies, gaining insights into structured concurrency and daemon fibers. Unravel the intricacies of when fibers run, enabling you to optimize their execution and harness the full potential of concurrent programming in Effect.

What is a Fiber?

A "fiber" is a small unit of work or a lightweight thread of execution. It represents a specific computation or an effectful operation in a program. Fibers are used to manage concurrency and asynchronous tasks.

Think of a fiber as a worker that performs a specific job. It can be started, paused, resumed, and even interrupted. Fibers are useful when we want to perform multiple tasks simultaneously or handle long-running operations without blocking the main program.

By using fibers, developers can control and coordinate the execution of tasks, allowing for efficient multitasking and responsiveness in their applications.

To summarize:

- An `Effect` is a higher-level concept that describes an effectful computation. It is lazy and immutable, meaning it represents a computation that may produce a value or fail but does not immediately execute.
- A fiber, on the other hand, represents the running execution of an `Effect`. It can be interrupted or awaited to retrieve its result. Think of it as a way to control and interact with the ongoing computation.

Creating Fibers

A fiber is created any time an effect is run. When running effects concurrently, a fiber is created for each concurrent effect.

Lifetime of Child Fibers

When we fork fibers, depending on how we fork them we can have four different lifetime strategies for the child fibers:

1. **Fork With Automatic Supervision**. If we use the ordinary `Effect.fork` operation, the child fiber will be automatically supervised by the parent fiber. The lifetime child fibers are tied to the lifetime of their parent fiber. This means that these fibers will be terminated either when they end naturally, or when their parent fiber is terminated.

2. **Fork in Global Scope (Daemon)**. Sometimes we want to run long-running background fibers that aren't tied to their parent fiber, and also we want to fork them in a global scope. Any fiber that is forked in global scope will become daemon fiber. This can be achieved by using the `Effect.forkDaemon` operator. As these fibers have no parent, they are not supervised, and they will be terminated when they end naturally, or when our application is terminated.

3. **Fork in Local Scope**. Sometimes, we want to run a background fiber that isn't tied to its parent fiber, but we want to live that fiber in the local scope. We can fork fibers in the local scope by using `Effect.forkScoped`. Such fibers can outlive their parent fiber (so they are not supervised by their parents), and they will be terminated when their life end or their local scope is closed.

4. **Fork in Specific Scope**. This is similar to the previous strategy, but we can have more fine-grained control over the lifetime of the child fiber by forking it in a specific scope. We can do this by using the `Effect.forkIn` operator.

Fork with Automatic Supervision

Effect employs a **structured concurrency** model where the lifetimes of fibers are neatly nested. Simply put, the lifespan of a fiber depends on the lifespan of its parent fiber.

To help clarify this concept, let's explore the following example. In this scenario, the `parent` fiber spawns the `child` fiber. The `child` fiber is engaged in a long-running task that never completes. What's important to note here is that Effect ensures the `child` fiber will not outlive the `parent` fiber:

```
import { Effect, Console, Schedule } from "effect"

const child = Effect.repeat(
  Console.log("child: still running!"),
  Schedule.fixed("1 second")
)

const parent = Effect.gen(function* () {
  console.log("parent: started!")
  yield* Effect.fork(child)
  yield* Effect.sleep("3 seconds")
```

```
  console.log("parent: finished!")
})

Effect.runPromise(parent)
```

When you run the above program, you'll see the following output:

```
parent: started!
child: still running!
child: still running!
child: still running!
parent: finished!
```

This pattern can be extended to any level of nested fibers.

Fork in Global Scope (Daemon)

Using `Effect.forkDaemon` we can create a daemon fiber from an effect. Its lifetime is tied to the global scope. So if the parent fiber terminates, the daemon fiber will not be terminated. It will only will be terminated when the global scope is closed, or its life end naturally.

```
import { Effect, Console, Schedule } from "effect"

const daemon = Effect.repeat(
  Console.log("daemon: still running!"),
  Schedule.fixed("1 second")
)

const parent = Effect.gen(function* () {
  console.log("parent: started!")
  yield* Effect.forkDaemon(daemon)
  yield* Effect.sleep("3 seconds")
  console.log("parent: finished!")
})

Effect.runPromise(parent)
```

If we run the above program, we will see the following output which shows that while the lifetime of the `parent` fiber ends after 3 seconds, the `daemon` fiber is still running:

```
parent: started!
daemon: still running!
daemon: still running!
daemon: still running!
parent: finished!
daemon: still running!
daemon: still running!
daemon: still running!
daemon: still running!
daemon: still running!
...etc...
```

Even if we interrupt the `parent` fiber, the `daemon` fiber will not be interrupted:

```
import { Effect, Console, Schedule, Fiber } from "effect"

const daemon = Effect.repeat(
  Console.log("daemon: still running!"),
  Schedule.fixed("1 second")
)

const parent = Effect.gen(function* () {
  console.log("parent: started!")
  yield* Effect.forkDaemon(daemon)
  yield* Effect.sleep("3 seconds")
  console.log("parent: finished!")
}).pipe(Effect.onInterrupt(() => Console.log("parent: interrupted!")))

const program = Effect.gen(function* () {
  const fiber = yield* Effect.fork(parent)
  yield* Effect.sleep("2 seconds")
  yield* Fiber.interrupt(fiber)
})

Effect.runPromise(program)
```

The output:

```
parent: started!
daemon: still running!
daemon: still running!
parent: interrupted!
daemon: still running!
daemon: still running!
daemon: still running!
daemon: still running!
daemon: still running!
...etc...
```

Fork in Local Scope

Sometimes we want to attach fiber to a local scope. In such cases, we can use the `Effect.forkScoped` operator. By using this operator, the lifetime of the forked fiber can be outlived the lifetime of its parent fiber, and it will be terminated when the local scope is closed:

```
import { Effect, Console, Schedule } from "effect"

const child = Effect.repeat(
  Console.log("child: still running!"),
  Schedule.fixed("1 second")
)

const parent = Effect.gen(function* () {
  console.log("parent: started!")
  yield* Effect.forkScoped(child)
  yield* Effect.sleep("3 seconds")
  console.log("parent: finished!")
})

const program = Effect.scoped(
  Effect.gen(function* () {
    console.log("Local scope started!")
    yield* Effect.fork(parent)
    yield* Effect.sleep("5 seconds")
    console.log("Leaving the local scope!")
  })
)

Effect.runPromise(program)
```

In the above example, the `child` fiber forked in the local scope has bigger lifetime than its `parent` fiber. So, when its `parent` fiber is terminated, the `child` fiber still running in the local scope until the local scope is closed. Let's see the output:

```
Local scope started!
parent: started!
child: still running!
child: still running!
child: still running!
parent: finished!
child: still running!
child: still running!
Leaving the local scope!
```

Fork in Specific Scope

There are some cases where we need more fine-grained control, so we want to fork a fiber in a specific scope. We can use the `Effect.forkIn` operator which takes the target scope as an argument:

```
import { Console, Effect, Schedule } from "effect"

const child = Console.log("child: still running!").pipe(
  Effect.repeat(Schedule.fixed("1 second"))
)

const program = Effect.scoped(
  Effect.gen(function* () {
    yield* Effect.addFinalizer(() =>
      Console.log("The outer scope is about to be closed!")
    )

    const outerScope = yield* Effect.scope

    yield* Effect.scoped(
      Effect.gen(function* () {
        yield* Effect.addFinalizer(() =>
          Console.log("The inner scope is about to be closed!")
        )
        yield* Effect.forkIn(child, outerScope)
        yield* Effect.sleep("3 seconds")
      })
    )

    yield* Effect.sleep("5 seconds")
  })
)

Effect.runPromise(program)
```

The output:

```
child: still running!
child: still running!
child: still running!
The inner scope is about to be closed!
child: still running!
child: still running!
child: still running!
```

```
child: still running!
child: still running!
child: still running!
The outer scope is about to be closed!
```

When do Fibers run?

New fibers begin execution after the current fiber completes or yields. This is necessary to prevent infinite loops in some cases, and is useful to know when using the `fork` APIs.

In the following example the `SubscriptionRef` `changes` stream only contains a single value `2` because the fiber (created by `fork`) to run the stream is started *after* the value has been updated.

```
import { Effect, SubscriptionRef, Stream, Console } from "effect"

const program = Effect.gen(function* () {
  const ref = yield* SubscriptionRef.make(0)
  yield* ref.changes.pipe(
    Stream.tap((n) => Console.log(`SubscriptionRef changed to ${n}`)),
    Stream.runDrain,
    Effect.fork
  )
  yield* SubscriptionRef.set(ref, 1)
  yield* SubscriptionRef.set(ref, 2)
})

Effect.runPromise(program)
/*
Output:
SubscriptionRef changed to 2
*/
```

If we add `Effect.yieldNow()` to force the current fiber to yield then the stream will contain all values `0`, `1`, and `2` because the fiber running the stream has an opportunity to start before the value is changed.

```
import { Effect, SubscriptionRef, Stream, Console } from "effect"

const program = Effect.gen(function* () {
  const ref = yield* SubscriptionRef.make(0)
  yield* ref.changes.pipe(
    Stream.tap((n) => Console.log(`SubscriptionRef changed to ${n}`)),
    Stream.runDrain,
    Effect.fork
  )
  yield* Effect.yieldNow()
  yield* SubscriptionRef.set(ref, 1)
  yield* SubscriptionRef.set(ref, 2)
})

Effect.runPromise(program)
/*
Output:
SubscriptionRef changed to 0
SubscriptionRef changed to 1
SubscriptionRef changed to 2
*/
```

Introduction to Runtime

Location: 400-guides/500-runtime

Effect is a powerful TypeScript library designed to help developers easily create complex, synchronous, and asynchronous programs.

The `Runtime<R>` data type represents a Runtime System that can execute effects. To execute any effect, we need a `Runtime` that includes the necessary requirements for that effect.

A `Runtime<R>` consists of three main components:

- a value of type `Context<R>`
- a value of type `FiberRefs`
- a value of type `RuntimeFlags`

The Default Runtime

When we use functions like `Effect.run*`, we are actually using the **default runtime** without explicitly mentioning it. These functions are designed as convenient shortcuts for executing our effects using the default runtime.

For instance, in the `Runtime` module, there is a corresponding `Runtime.run*(defaultRuntime)` function that is called internally by `Effect.run*`, e.g. `Effect.runSync` is simply an alias for `Runtime.runSync(defaultRuntime)`.

The default runtime includes:

- An empty `Context<never>`
- A set of `FiberRefs` that include the default services
- A default configuration for `RuntimeFlags` that enables `Interruption` and `CooperativeYielding`

In most cases, using the default runtime is sufficient. However, it can be useful to create a custom runtime to reuse a specific context or configuration. It is common to create a `Runtime<R>` by initializing a `Layer<R, Err, RIn>`. This allows for context reuse across execution boundaries, such as in a React App or when executing operations on a server in response to API requests.

What is a Runtime System?

When we write an Effect program, we construct an `Effect` using constructors and combinators. Essentially, we are creating a blueprint of a program. An `Effect` is merely a data structure that describes the execution of a concurrent program. It represents a tree-like structure that combines various primitives to define what the `Effect` should do.

However, this data structure itself does not perform any actions; it is solely a description of a concurrent program.

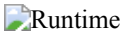
Therefore, it is crucial to understand that when working with a functional effect system like Effect, our code for actions such as printing to the console, reading files, or querying databases is actually building a workflow or blueprint for an application. We are constructing a data structure.

So how does Effect actually run these workflows? This is where the Effect Runtime System comes into play. When we invoke a `Runtime.run*` function, the Runtime System takes over. First, it creates an empty root Fiber with:

- The initial context
- The initial fiberRefs
- The initial Effect

After the creation of the Fiber, it invokes the Fiber's runLoop, which follows the instructions described by the `Effect` and executes them step by step.

To simplify, we can envision the Runtime System as a black box that takes both the effect Effect<A, E, R> and its associated context `Context<R>`. It runs the effect and returns the result as an Exit<A, E> value.



Responsibilities of the Runtime System

Runtime Systems have a lot of responsibilities:

1. **Execute every step of the blueprint**. They have to execute every step of the blueprint in a while loop until it's done.
2. **Handle unexpected errors**. They have to handle unexpected errors, not just the expected ones but also the unexpected ones.
3. **Spawn concurrent fiber**. They are actually responsible for the concurrency that effect systems have. They have to spawn a fiber every time we call `fork` on an effect to spawn off a new fiber.
4. **Cooperatively yield to other fibers**. They have to cooperatively yield to other fibers so that fibers that are sort of hogging the spotlight, don't get to monopolize all the CPU resources.
5. **Ensure finalizers are run appropriately**. They have to ensure finalizers are run appropriately at the right point in all circumstances to make sure that resources are closed that clean-up logic is executed. This is the feature that powers Scope and all the other resource-safe constructs in Effect.
6. **Handle asynchronous callback**. They have to handle this messy job of dealing with asynchronous callbacks. So we don't have to deal with async code. When we are using Effect, everything can be interpreted as async or sync out of the box.

Default Runtime

Effect provides a default runtime named `Runtime.defaultRuntime` designed for mainstream usage.

The default runtime provides the minimum capabilities to bootstrap execution of Effect tasks.

Both of the following executions are equivalent:

```
import { Effect, Runtime } from "effect"

const program = Effect.log("Application started!")

Effect.runSync(program)
/*
Output:
... level=INFO fiber=#0 message="Application started!"
*/

Runtime.runSync(Runtime.defaultRuntime)(program)
/*
Output:
... level=INFO fiber=#0 message="Application started!"
*/
```

Under the hood, `Effect.runSync` (and the same principle applies to other `Effect.run*` functions) serves as a convenient shorthand for `Runtime.runSync(Runtime.defaultRuntime)`.

Locally Scoped Runtime Configuration

In Effect, runtime configurations are typically inherited from their parent workflows. This means that when we access a runtime configuration or obtain a runtime inside a workflow, we are essentially using the configuration of the parent workflow. However, there are cases where we want to temporarily override the runtime configuration for a specific part of our code. This concept is known as locally scoped runtime configuration. Once the execution of that code region is completed, the runtime configuration reverts to its original settings.

To achieve this, we make use of `Effect.provide*` functions, which allow us to provide a new runtime configuration to a specific section of our code.

Configuring Runtime by Providing Configuration Layers

By utilizing the `Effect.provide` function and providing runtime configuration layers to an Effect workflow, we can easily modify runtime configurations.

Here's an example:

```
import { Logger, Effect } from "effect"

// Define a configuration layer
const addSimpleLogger = Logger.replace(
  Logger.defaultLogger,
  Logger.make(({ message }) => console.log(message))
)

const program = Effect.gen(function* () {
  yield* Effect.log("Application started!")
  yield* Effect.log("Application is about to exit!")
})

Effect.runSync(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message="Application started!"
timestamp=... level=INFO fiber=#0 message="Application is about to exit!"
*/

// Overriding the default logger
Effect.runSync(program.pipe(Effect.provide(addSimpleLogger)))
/*
Output:
Application started!
Application is about to exit!
*/
```

In this example, we first create a configuration layer for a simple logger using `Logger.replace`. Then, we use `Effect.provide` to provide this configuration to our program, effectively overriding the default logger with the simple logger.

To ensure that the runtime configuration is only applied to a specific part of an Effect application, we should provide the configuration layer exclusively to that particular section, as demonstrated in the following example:

```
import { Logger, Effect } from "effect"

// Define a configuration layer
const addSimpleLogger = Logger.replace(
  Logger.defaultLogger,
  Logger.make(({ message }) => console.log(message))
)

const program = Effect.gen(function* () {
  yield* Effect.log("Application started!")
  yield* Effect.gen(function* () {
    yield* Effect.log("I'm not going to be logged!")
    yield* Effect.log("I will be logged by the simple logger.").pipe(
      Effect.provide(addSimpleLogger)
    )
    yield* Effect.log(
      "Reset back to the previous configuration, so I won't be logged."
    )
  }).pipe(Effect.provide(Logger.remove(Logger.defaultLogger)))
  yield* Effect.log("Application is about to exit!")
})

Effect.runSync(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message="Application started!"
I will be logged by the simple logger.
timestamp=... level=INFO fiber=#0 message="Application is about to exit!"
*/
```

Top-level Runtime Configuration

When developing an Effect application and executing it using `Effect.run*` functions, the application is automatically run using the default runtime behind the scenes. While we can adjust and customize specific aspects of our Effect application by providing locally scoped configuration layers using `Effect.provide` operations, there are scenarios where we need to customize the runtime configuration for the entire application from the top level.

In such situations, we can create a top-level runtime by converting a configuration layer into a runtime using the `ManagedRuntime.make` constructor.

ManagedRuntime

```
import { Effect, ManagedRuntime, Logger } from "effect"

// Define a configuration layer
const appLayer = Logger.replace(
  Logger.defaultLogger,
  Logger.make(({ message }) => console.log(message))
)

// Transform the configuration layer into a runtime
const runtime = ManagedRuntime.make(appLayer)

const program = Effect.log("Application started!")

// Execute the program using the custom runtime
runtime.runSync(program)

// Cleaning up any resources used by the configuration layer
Effect.runFork(runtime.disposeEffect)
/*
Output:
Application started!
*/
```

In this example, we first create a custom configuration layer called `appLayer`, which includes modifications to the logger configuration. Next, we transform this configuration layer into a runtime using `ManagedRuntime.make`. This results in a top-level runtime that encapsulates the desired configuration for the entire Effect application.

By customizing the top-level runtime configuration, we can tailor the behavior of our entire Effect application to meet our specific needs and requirements.

Effect.Tag

When you utilize a runtime that you pass around, you can use `Effect.Tag` to define a new tag and simplify access to a service. This incorporates the service shape directly into the static side of the tag class.

You can define a new tag using `Effect.Tag` as shown below:

```
import { Effect } from "effect"

class Notifications extends Effect.Tag("Notifications")<
  Notifications,
  { readonly notify: (message: string) => Effect.Effect<void> }
>() {}
```

In this setup, every field of the service shape is converted into a static property of the `Notifications` class.

This allows you to access the service shape directly:

```
import { Effect } from "effect"

class Notifications extends Effect.Tag("Notifications")<
  Notifications,
  { readonly notify: (message: string) => Effect.Effect<void> }
>() {}

// ---cut---
const action = Notifications.notify("Hello, world!")
```

As you can see, `action` depends on `Notifications`, but this isn't a problem because you can later construct a `Layer` that provides `Notifications` and build a `ManagedRuntime` with it.

Integrations

The `ManagedRuntime` simplifies the integration of services and layers with other frameworks or tools, particularly in environments where Effect is not the primary framework and access to the main entry point is restricted.

For instance, `ManagedRuntime` can be particularly useful in environments like React or other frameworks where control over the main application entry point is limited. Here's how you can use `ManagedRuntime` to manage service lifecycle within an external framework:

```
import { Effect, ManagedRuntime, Layer, Console } from "effect"
```

```
class Notifications extends Effect.Tag("Notifications")<
  Notifications,
  { readonly notify: (message: string) => Effect.Effect<void> }
>() {
  static Live = Layer.succeed(this, {
    notify: (message) => Console.log(message)
  })
}

// Example entry point for an external framework
async function main() {
  const runtime = ManagedRuntime.make(Notifications.Live)
  await runtime.runPromise(Notifications.notify("Hello, world!"))
  await runtime.dispose()
}
```

Configuration

Location: 400-guides/450-configuration

Effect is a powerful TypeScript library designed to help developers easily create complex, synchronous, and asynchronous programs.

Configuration is an essential aspect of any cloud-native application. Effect simplifies the process of managing configuration by offering a convenient interface for configuration providers.

The configuration front-end in Effect enables ecosystem libraries and applications to specify their configuration requirements in a declarative manner. It offloads the complex tasks to a `ConfigProvider`, which can be supplied by third-party libraries.

Effect comes bundled with a straightforward default `ConfigProvider` that retrieves configuration data from environment variables. This default provider can be used during development or as a starting point before transitioning to more advanced configuration providers.

To make our application configurable, we need to understand three essential elements:

- **Config Description**: We describe the configuration data using an instance of `Config<A>`. If the configuration data is simple, such as a `string`, `number`, or `boolean`, we can use the built-in functions provided by the `Config` module. For more complex data types like HostPort, we can combine primitive configs to create a custom configuration description.
- **Config Frontend**: We utilize the instance of `Config<A>` to load the configuration data described by the instance (a `Config` is, in itself, an effect). This process leverages the current `ConfigProvider` to retrieve the configuration.
- **Config Backend**: The `ConfigProvider` serves as the underlying engine that manages the configuration loading process. Effect comes with a default config provider as part of its default services. This default provider reads the configuration data from environment variables. If we want to use a custom config provider, we can utilize the `Layer.setConfigProvider` layer to configure the Effect runtime accordingly.

Getting Started

Effect provides a set of primitives for the most common types like `string`, `number`, `boolean`, `integer`, etc.

Let's start with a simple example of how to read configuration from environment variables:

If we run this program we will get the following output:

```
npx ts-node primitives.ts
(Missing data at HOST: "Expected HOST to exist in the process context")
```

This is because we have not provided any configuration. Let's try running it with the following environment variables:

```
HOST=localhost PORT=8080 npx ts-node primitives.ts
Application started: localhost:8080
```

Primitives

Effect offers these basic types out of the box:

- `string`: Constructs a config for a string value.
- `number`: Constructs a config for a float value.
- `boolean`: Constructs a config for a boolean value.
- `integer`: Constructs a config for a integer value.
- `date`: Constructs a config for a date value.
- `literal`: Constructs a config for a literal (*) value.
- `logLevel`: Constructs a config for a LogLevel value.
- `duration`: Constructs a config for a duration value.
- `redacted`: Constructs a config for a secret value.

(*) `string | number | boolean | null | bigint`

Default Values

In some cases, you may encounter situations where an environment variable is not set, leading to a missing value in the configuration. To handle such scenarios, Effect provides the `Config.withDefault` function. This function allows you to specify a fallback or default value to use when an environment variable is not present.

Here's how you can use `Config.withDefault` to handle fallback values:

When running the program with the command:

```
HOST=localhost npx ts-node withDefault.ts
```

you will see the following output:

```
Application started: localhost:8080
```

Even though the `PORT` environment variable is not set, the fallback value of `8080` is used, ensuring that the program continues to run smoothly with a default value.

Constructors

Effect provides several built-in constructors. These are functions that take a `Config` as input and produce another `Config`.

- `array`: Constructs a config for an array of values.
- `chunk`: Constructs a config for a sequence of values.
- `option`: Returns an optional version of this config, which will be `None` if the data is missing from configuration, and `Some` otherwise.
- `repeat`: Returns a config that describes a sequence of values, each of which has the structure of this config.
- `hashSet`: Constructs a config for a sequence of values.
- `hashMap`: Constructs a config for a sequence of values.

In addition to the basic ones, there are three special constructors you might find useful:

- `succeed`: Constructs a config which contains the specified value.
- `fail`: Constructs a config that fails with the specified message.
- `all`: Constructs a config from a tuple / struct / arguments of configs.

Example

```
import { Effect, Config } from "effect"

const program = Effect.gen(function* () {
  const config = yield* Config.array(Config.string(), "MY_ARRAY")
  console.log(config)
})

Effect.runSync(program)

MY_ARRAY=a,b,c npx ts-node array.ts

[ 'a', 'b', 'c' ]
```

Operators

Effect comes with a set of built-in operators to help you manipulate and handle configurations.

Transforming Operators

These operators allow you to transform a config into a new one:

- `validate`: Returns a config that describes the same structure as this one, but which performs validation during loading.
- `map`: Creates a new config with the same structure as the original but with values transformed using a given function.
- `mapAttempt`: Similar to `map`, but if the function throws an error, it's caught and turned into a validation error.
- `mapOrFail`: Like `map`, but allows for functions that might fail. If the function fails, it results in a validation error.

Example

```
import { Effect, Config } from "effect"

const program = Effect.gen(function* () {
  const config = yield* Config.string("NAME").pipe(
    Config.validate({
      message: "Expected a string at least 4 characters long",
      validation: (s) => s.length >= 4
    })
  )
  console.log(config)
})

Effect.runSync(program)

NAME=foo npx ts-node validate.ts

[(Invalid data at NAME: "Expected a string at least 4 characters long")]
```

Fallback Operators

These operators help you set up fallbacks in case of errors or missing data:

- `orElse`: Sets up a config that tries to use this config first. If there's an issue, it falls back to another specified config.
- `orElseIf`: This one also tries to use the main config first but switches to a fallback config if there's an error that matches a specific condition.

Example

In this example, we have a program that requires two configurations: `A` and `B`. We will use two configuration providers, where each provider has only one of the configurations. We will demonstrate how to set up fallbacks using the `orElse` operator.

```
import { Config, ConfigProvider, Effect, Layer } from "effect"

// A program that requires two configurations: A and B
const program = Effect.gen(function* () {
  const A = yield* Config.string("A")
  const B = yield* Config.string("B")
  console.log(`A: ${A}`, `B: ${B}`)
})

const provider1 = ConfigProvider.fromMap(
  new Map([
    ["A", "A"]
    // B is missing
  ])
)

const provider2 = ConfigProvider.fromMap(
  new Map([
    // A is missing
    ["B", "B"]
  ])
)

const layer = Layer.setConfigProvider(
  provider1.pipe(ConfigProvider.orElse(() => provider2))
)

Effect.runSync(Effect.provide(program, layer))

ts-node npx orElse.ts

A: A B: B
```

<Info> The `ConfigProvider.fromMap` method used in this example creates a configuration provider from a `Map`. This behavior is explained in more detail in the Testing Services section. </Info>

Custom Configurations

In addition to primitive types, we can also define configurations for custom types. To achieve this, we use primitive configs and combine them using `Config` operators (`zip`, `orElse`, `map`, etc.) and constructors (`array`, `hashSet`, etc.).

Let's consider the `HostPort` data type, which consists of two fields: `host` and `port`.

```
class HostPort {
  constructor(
    readonly host: string,
    readonly port: number
  ) {}
}
```

We can define a configuration for this data type by combining primitive configs for `string` and `number`:

```
import { Config } from "effect"

export class HostPort {
  constructor(
    readonly host: string,
    readonly port: number
  ) {}

  get url() {
    return `${this.host}:${this.port}`
  }
}

const both = Config.all([Config.string("HOST"), Config.number("PORT")])

export const config = Config.map(
  both,
  ([host, port]) => new HostPort(host, port)
)
```

In the above example, we use the `Config.all(configs)` operator to combine two primitive configs `Config<string>` and `Config<number>` into a `Config<[string, number]>`.

If we use this customized configuration in our application:

when you run the program using `Effect.runSync(program)`, it will attempt to read the corresponding values from environment variables (`HOST` and `PORT`):

```
HOST=localhost PORT=8080 npx ts-node HostPort.ts

Application started: localhost:8080
```

Top-level and Nested Configurations

So far, we have learned how to define configurations in a top-level manner, whether they are for primitive or custom types. However, we can also define nested configurations.

Let's assume we have a `ServiceConfig` data type that consists of two fields: `hostPort` and `timeout`.

```
// @filename: HostPort.ts
import { Config } from "effect"

export class HostPort {
  constructor(
    readonly host: string,
    readonly port: number
  ) {}

  get url() {
    return `${this.host}:${this.port}`
  }
}

const both = Config.all([Config.string("HOST"), Config.number("PORT")])

export const config = Config.map(
  both,
  ([host, port]) => new HostPort(host, port)
)

// @filename: ServiceConfig.ts
// ---cut---
import * as HostPort from "./HostPort"
import { Config } from "effect"

class ServiceConfig {
  constructor(
    readonly hostPort: HostPort.HostPort,
    readonly timeout: number
  ) {}
}

const config = Config.map(
  Config.all([HostPort.config, Config.number("TIMEOUT")]),
  ([hostPort, timeout]) => new ServiceConfig(hostPort, timeout)
)
```

If we use this customized config in our application, it tries to read corresponding values from environment variables: `HOST`, `PORT`, and `TIMEOUT`.

However, in many cases, we don't want to read all configurations from the top-level namespace. Instead, we may want to nest them under a common namespace. For example, we want to read both `HOST` and `PORT` from the `HOSTPORT` namespace, and `TIMEOUT` from the root namespace.

To achieve this, we can use the `Config.nested` combinator. It allows us to nest configs under a specific namespace. Here's how we can update our configuration:

```
// @filename: HostPort.ts
import { Config } from "effect"

export class HostPort {
  constructor(
    readonly host: string,
    readonly port: number
  ) {}

  get url() {
    return `${this.host}:${this.port}`
  }
}

const both = Config.all([Config.string("HOST"), Config.number("PORT")])

export const config = Config.map(
  both,
  ([host, port]) => new HostPort(host, port)
)
```

```
// @filename: ServiceConfig.ts
import * as HostPort from "./HostPort"
import { Config } from "effect"

class ServiceConfig {
  constructor(
    readonly hostPort: HostPort.HostPort,
    readonly timeout: number
  ) {}
}

// ---cut---
const config = Config.map(
  Config.all([
    Config.nested(HostPort.config, "HOSTPORT"),
    Config.number("TIMEOUT")
  ]),
  ([hostPort, timeout]) => new ServiceConfig(hostPort, timeout)
)
```

Now, if we run our application, it will attempt to read the corresponding values from the environment variables: `HOSTPORT_HOST`, `HOSTPORT_PORT`, and `TIMEOUT`.

Testing Services

When testing services, there are scenarios where we need to provide specific configurations to them. In such cases, we should be able to mock the backend that reads the configuration data.

To accomplish this, we can use the `ConfigProvider.fromMap` constructor. This constructor takes a `Map<string, string>` that represents the configuration data, and it returns a config provider that reads the configuration from that map.

Once we have the mock config provider, we can use `Layer.setConfigProvider` function. This function allows us to override the default config provider and provide our own custom config provider. It returns a `Layer` that can be used to configure the Effect runtime for our test specs.

Here's an example of how we can mock a config provider for testing purposes:

```
// @filename: HostPort.ts
import { Config } from "effect"

export class HostPort {
  constructor(
    readonly host: string,
    readonly port: number
  ) {}

  get url() {
    return `${this.host}:${this.port}`
  }
}

const both = Config.all([Config.string("HOST"), Config.number("PORT")])

export const config = Config.map(
  both,
  ([host, port]) => new HostPort(host, port)
)

// @filename: App.ts
import { Effect, Console } from "effect"
import * as HostPort from "./HostPort"

export const program = HostPort.config.pipe(
  Effect.andThen((hostPort) =>
    Console.log(`Application started: ${hostPort.url}`)
  )
)

// @filename: mockConfigProvider.ts
// ---cut---
import { ConfigProvider, Layer, Effect } from "effect"
import * as App from "./App"

// Create a mock config provider using ConfigProvider.fromMap
const mockConfigProvider = ConfigProvider.fromMap(
  new Map([
    ["HOST", "localhost"],
    ["PORT", "8080"]
  ])
)

// Create a layer using Layer.setConfigProvider to override the default config provider
const layer = Layer.setConfigProvider(mockConfigProvider)

// Run the program using the provided layer
Effect.runSync(Effect.provide(App.program, layer))
// Output: Application started: localhost:8080
```

By using this approach, we can easily mock the configuration data and test our services with different configurations in a controlled manner.

Redacted

What sets `Config.redacted` apart from `Config.string` is its handling of sensitive information. It parses the config value and wraps it in a `Redacted<string>`, a data type designed for holding secrets.

When you use `Console.log` on a redacted, the actual value remains hidden, providing an added layer of security. The only way to access the value is by using `Redacted.value`.

```
import { Effect, Config, Console, Redacted } from "effect"

const program = Config.redacted("API_KEY").pipe(
  Effect.tap((redacted) => Console.log(`Console output: ${redacted}`)),
  Effect.tap((redacted) =>
    Console.log(`Actual value: ${Redacted.value(redacted)}`)
  )
)

Effect.runSync(program)
```

If we run this program we will get the following output:

```
API_KEY=my-api-key tsx Redacted.ts
Console output: <redacted>
Actual value: my-api-key
```

In this example, you can see that when logging the redacted using `Console.log`, the actual value is replaced with `<redacted>`, ensuring that sensitive information is not exposed. The `Redacted.value` function, on the other hand, provides a controlled way to retrieve the original secret value.

Secret

<Warning> Deprecated since version 3.3.0: Please use Config.redacted for handling sensitive information going forward. </Warning>

`Config.secret` functions similarly to `Config.redacted` by securing sensitive information. It wraps configuration values in a `Secret` type, which also obscures details when logged but allows access through the `Secret.value` method.

```
import { Effect, Config, Console, Secret } from "effect"

const program = Config.secret("API_KEY").pipe(
  Effect.tap((secret) => Console.log(`Console output: ${secret}`)),
  Effect.tap((secret) => Console.log(`Secret value: ${Secret.value(secret)}`))
)

Effect.runSync(program)
```

If we run this program we will get the following output:

```
API_KEY=my-api-key tsx Secret.ts
Console output: Secret(<redacted>)
Secret value: my-api-key
```

Patterns

Location: 400-guides/250-resource-management/200-patterns/index

Resource Management Patterns

Sequence of Operations with Compensating Actions on Failure

Location: 400-guides/250-resource-management/200-patterns/100-sequence-of-operations-with-compensating-actions-on-failure

Effect facilitates sequential operations where success depends on prior steps, with rollback on failure.

In certain scenarios, you might need to perform a sequence of chained operations where the success of each operation depends on the previous one. However, if any of the operations fail, you would want to reverse the effects of all previous successful operations. This pattern is valuable when you need to ensure that either all operations succeed, or none of them have any effect at all.

Effect offers a way to achieve this pattern using the Effect.acquireRelease function in combination with the Exit type. The Effect.acquireRelease function allows you to acquire a resource, perform operations with it, and release the resource when you're done. The Exit type represents the outcome of an effectful computation, indicating whether it succeeded or failed.

Let's go through an example of implementing this pattern. Suppose we want to create a "Workspace" in our application, which involves creating an S3 bucket, an ElasticSearch index, and a Database entry that relies on the previous two.

To begin, we define the domain model for the required services: `S3`, `ElasticSearch`, and `Database`.

```
import { Effect, Context } from "effect"

export class S3Error {
  readonly _tag = "S3Error"
}

export interface Bucket {
  readonly name: string
}

export class S3 extends Context.Tag("S3")<
  S3,
  {
    readonly createBucket: Effect.Effect<Bucket, S3Error>
    readonly deleteBucket: (bucket: Bucket) => Effect.Effect<void>
  }
>() {}

export class ElasticSearchError {
  readonly _tag = "ElasticSearchError"
}

export interface Index {
  readonly id: string
}

export class ElasticSearch extends Context.Tag("ElasticSearch")<
  ElasticSearch,
  {
    readonly createIndex: Effect.Effect<Index, ElasticSearchError>
    readonly deleteIndex: (index: Index) => Effect.Effect<void>
  }
>() {}

export class DatabaseError {
  readonly _tag = "DatabaseError"
}

export interface Entry {
  readonly id: string
}

export class Database extends Context.Tag("Database")<
  Database,
  {
    readonly createEntry: (
      bucket: Bucket,
      index: Index
    ) => Effect.Effect<Entry, DatabaseError>
    readonly deleteEntry: (entry: Entry) => Effect.Effect<void>
  }
>() {}

// @include: Services
```

Next, we define the three create actions and the overall transaction (`make`) for the Workspace.

We then create simple service implementations to test the behavior of our Workspace code. To achieve this, we will utilize layers to construct test services. These layers will be able to handle various scenarios, including errors, which we can control using the `FailureCase` type.

Let's examine the test results for the scenario where `FailureCase` is set to `undefined` (happy path):

```
[S3] creating bucket
[ElasticSearch] creating index
[Database] creating entry for bucket <bucket.name> and index <index.id>
{
  _id: "Either",
  _tag: "Right",
  right: {
    id: "<entry.id>"
  }
}
```

In this case, all operations succeed, and we see a successful result with `right({ id: '<entry.id>' })`.

Now, let's simulate a failure in the `Database`:

```
const runnable = Workspace.make.pipe(
  Effect.provide(layer),
  Effect.provideService(FailureCase, "Database")
)
```

The console output will be:

```
[S3] creating bucket
[ElasticSearch] creating index
[Database] creating entry for bucket <bucket.name> and index <index.id>
[ElasticSearch] delete index <index.id>
[S3] delete bucket <bucket.name>
```

```
{
  _id: "Either",
  _tag: "Left",
  left: {
    _tag: "DatabaseError"
  }
}
```

You can observe that once the `Database` error occurs, there is a complete rollback that deletes the `ElasticSearch` index first and then the associated `S3` bucket. The result is a failure with `left(new DatabaseError())`.

Let's now make the index creation fail instead:

```
const runnable = Workspace.make.pipe(
  Effect.provide(layer),
  Effect.provideService(FailureCase, "ElasticSearch")
)
```

In this case, the console output will be:

```
[S3] creating bucket
[ElasticSearch] creating index
[S3] delete bucket <bucket.name>
{
  _id: "Either",
  _tag: "Left",
  left: {
    _tag: "ElasticSearchError"
  }
}
```

As expected, once the `ElasticSearch` index creation fails, there is a rollback that deletes the `S3` bucket. The result is a failure with `left(new ElasticSearchError())`.

Resource Management

Location: 400-guides/250-resource-management/index

Resource Management

Scope

Location: 400-guides/250-resource-management/100-scope

Explore the importance of resource management in developing large-scale applications using the Effect library. Learn about robust constructs, such as the `Scope` data type, and discover how Effect simplifies resource management and ensures safety in your applications.

In the context of developing long-lived applications, resource management plays a critical role. Effective resource management is indispensable when building large-scale applications. It's imperative that our application is resource-efficient and does not result in resource leaks.

Resource leaks, such as unclosed socket connections, database connections, or file descriptors, are unacceptable in web applications. Effect offers robust constructs to address this concern effectively.

To create an application that manages resources safely, we must ensure that every time we open a resource, we have a mechanism in place to close it. This applies whether we fully utilize the resource or encounter exceptions during its use.

In the following sections, we'll delve deeper into how Effect simplifies resource management and ensures resource safety in your applications.

Scope

The `Scope` data type is fundamental for managing resources safely and in a composable manner in Effect.

In simple terms, a scope represents the lifetime of one or more resources. When a scope is closed, the resources associated with it are guaranteed to be released.

With the `Scope` data type, you can:

- **Add finalizers**, which represent the release of a resource.
- **Close** the scope, releasing all acquired resources and executing any added finalizers.

To grasp the concept better, let's delve into an example that demonstrates how it works. It's worth noting that in typical Effect usage, you won't often find yourself working with these low-level APIs for managing scopes.

```
import { Scope, Effect, Console, Exit } from "effect"

const program =
  // create a new scope
  Scope.make().pipe(
```

```
    // add finalizer 1
    Effect.tap((scope) =>
      Scope.addFinalizer(scope, Console.log("finalizer 1"))
    ),
    // add finalizer 2
    Effect.tap((scope) =>
      Scope.addFinalizer(scope, Console.log("finalizer 2"))
    ),
    // close the scope
    Effect.andThen((scope) =>
      Scope.close(scope, Exit.succeed("scope closed successfully"))
    )
  )

Effect.runPromise(program)
/*
Output:
finalizer 2 <-- finalizers are closed in reverse order
finalizer 1
*/
```

By default, when a `Scope` is closed, all finalizers added to that `Scope` are executed in the reverse order in which they were added. This approach makes sense because releasing resources in the reverse order of acquisition ensures that resources are properly closed.

For instance, if you open a network connection and then access a file on a remote server, you must close the file before closing the network connection. This sequence is critical to maintaining the ability to interact with the remote server.

By combining `Scope` with the Effect context, we gain a powerful way to manage resources effectively.

addFinalizer

Now, let's dive into the `Effect.addFinalizer` function, which provides a higher-level API for adding finalizers to the scope of an `Effect` value. These finalizers are guaranteed to execute when the associated scope is closed, and their behavior may depend on the `Exit` value with which the scope is closed.

Let's explore some examples to understand this better.

Let's observe how things behave in the event of success:

Here, the `Effect.addFinalizer` operator adds a `Scope` to the context required by the workflow, as indicated by:

```
Effect<void, never, Scope>
```

This signifies that the workflow needs a `Scope` to execute. You can provide this `Scope` using the `Effect.scoped` operator. It creates a new `Scope`, supplies it to the workflow, and closes the `Scope` once the workflow is complete.

The `Effect.scoped` operator removes the `Scope` from the context, indicating that the workflow no longer requires any resources associated with a scope.

Next, let's explore how things behave in the event of a failure:

Finally, let's explore the behavior in the event of an interruption:

Manually Create and Close Scopes

When you're working with multiple scoped resources within a single operation, it's important to understand how their scopes interact. By default, these scopes are merged into one, but you can have more fine-grained control over when each scope is closed by manually creating and closing them.

Let's start by looking at how scopes are merged by default. Take a look at this code example:

In this case, the scopes of `task1` and `task2` are merged into a single scope, and when the program is run, it outputs the tasks and their finalizers in a specific order.

If you want more control over when each scope is closed, you can manually create and close them, as shown in this example:

In this example, we create two separate scopes, `scope1` and `scope2`, and extend the scope of each task into its respective scope. When you run the program, it outputs the tasks and their finalizers in a different order.

<Info> The `Scope.extend` function allows you to extend the scope of an `Effect` workflow that requires a scope into another scope without closing the scope when the workflow finishes executing. This allows you to extend a scoped value into a larger scope. </Info>

You might wonder what happens when a scope is closed, but a task within that scope hasn't completed yet. The key point to note is that the scope closing doesn't force the task to be interrupted. It will continue running, and the finalizer will execute immediately when registered. The call to `close` will only wait for the finalizers that have already been registered.

So, finalizers run when the scope is closed, not necessarily when the effect finishes running. When you're using `Effect.scoped`, the scope is managed automatically, and finalizers are executed accordingly. However, when you manage the scope manually, you have control over when finalizers are executed.

Defining Resources

We can define a resource using operators like `Effect.acquireRelease(acquire, release)`, which allows us to create a scoped value from an `acquire` and `release` workflow.

Every acquire release requires three actions:

- **Acquiring Resource**. An effect describing the acquisition of resource. For example, opening a file.
- **Using Resource**. An effect describing the actual process to produce a result. For example, counting the number of lines in a file.
- **Releasing Resource**. An effect describing the final step of releasing or cleaning up the resource. For example, closing a file.

The `Effect.acquireRelease` operator performs the `acquire` workflow **uninterruptibly**. This is important because if we allowed interruption during resource acquisition we could be interrupted when the resource was partially acquired.

The guarantee of the `Effect.acquireRelease` operator is that if the `acquire` workflow successfully completes execution then the `release` workflow is guaranteed to be run when the `Scope` is closed.

For example, let's define a simple resource:

```
import { Effect } from "effect"

// Define the interface for the resource
export interface MyResource {
  readonly contents: string
  readonly close: () => Promise<void>
}

// Simulate getting the resource
const getMyResource = (): Promise<MyResource> =>
  Promise.resolve({
    contents: "lorem ipsum",
    close: () =>
      new Promise((resolve) => {
        console.log("Resource released")
        resolve()
      })
  })

// Define the acquisition of the resource with error handling
export const acquire = Effect.tryPromise({
  try: () =>
    getMyResource().then((res) => {
      console.log("Resource acquired")
      return res
    }),
  catch: () => new Error("getMyResourceError")
})

// Define the release of the resource
export const release = (res: MyResource) => Effect.promise(() => res.close())

export const resource = Effect.acquireRelease(acquire, release)

// @include: resource
```

Notice that the `Effect.acquireRelease` operator added a `Scope` to the context required by the workflow:

```
Effect<MyResource, Error, Scope>
```

This indicates that the workflow needs a `Scope` to run and adds a finalizer that will close the resource when the scope is closed.

We can continue working with the resource for as long as we want by using `Effect.andThen` or other Effect operators. For example, here's how we can read the contents:

Once we are done working with the resource, we can close the scope using the `Effect.scoped` operator. It creates a new `Scope`, provides it to the workflow, and closes the `Scope` when the workflow is finished.

The `Effect.scoped` operator removes the `Scope` from the context, indicating that there are no longer any resources used by this workflow which require a scope.

We now have a workflow that is ready to run:

```
Effect.runPromise(program)
/*
Resource acquired
content is lorem ipsum
Resource released
*/
```

acquireUseRelease

The `Effect.acquireUseRelease(acquire, use, release)` function is a specialized version of the `Effect.acquireRelease` function that simplifies resource management by automatically handling the scoping of resources.

The main difference is that `acquireUseRelease` eliminates the need to manually call `Effect.scoped` to manage the resource's scope. It has additional knowledge about when you are done using the resource created with the `acquire` step. This is achieved by providing a `use` argument, which represents the function that operates on the acquired resource. As a result, `acquireUseRelease` can automatically determine when it should execute the release step.

Here's an example that demonstrates the usage of `acquireUseRelease`:

```
// @filename: resource.ts
// @include: resource

// @filename: index.ts
// ---cut---
import { Effect, Console } from "effect"
import { MyResource, acquire, release } from "./resource"

const use = (res: MyResource) => Console.log(`content is ${res.contents}`)

const program = Effect.acquireUseRelease(acquire, use, release)

Effect.runPromise(program)
/*
Output:
Resource acquired
content is lorem ipsum
Resource released
*/
```

Fallback

Location: 400-guides/150-error-management/400-fallback

Explore techniques for handling failures and fallbacks in Effect, including `orElse` to try alternate effects, `orElseFail` and `orElseSucceed` to modify failures, and `firstSuccessOf` to retrieve the result of the first successful effect. Learn how to gracefully handle errors and create fallback mechanisms in your Effect programs.

orElse

We can attempt one effect, and if it fails, we can try another effect using the `Effect.orElse` combinator:

```
import { Effect } from "effect"

const success = Effect.succeed("success")
const failure = Effect.fail("failure")
const fallback = Effect.succeed("fallback")

const program1 = Effect.orElse(success, () => fallback)
console.log(Effect.runSync(program1)) // Output: "success"

const program2 = Effect.orElse(failure, () => fallback)
console.log(Effect.runSync(program2)) // Output: "fallback"
```

orElseFail / orElseSucceed

These two operators modify the original failure by replacing it with constant succeed or failure values.

The `Effect.orElseFail` will always replace the original failure with the new one:

```
import { Effect } from "effect"

class NegativeAgeError {
  readonly _tag = "NegativeAgeError"
  constructor(readonly age: number) {}
}

class IllegalAgeError {
  readonly _tag = "IllegalAgeError"
  constructor(readonly age: number) {}
}

const validate = (
  age: number
): Effect.Effect<number, NegativeAgeError | IllegalAgeError> => {
  if (age < 0) {
    return Effect.fail(new NegativeAgeError(age))
  } else if (age < 18) {
    return Effect.fail(new IllegalAgeError(age))
  } else {
    return Effect.succeed(age)
  }
}

// @include: validate
```

```
const program1 = Effect.orElseFail(validate(3), () => "invalid age")
```

The `Effect.orElseSucceed` will always replace the original failure with a success value, so the resulting effect cannot fail:

```
// @include: validate
// ---cut---
const program2 = Effect.orElseSucceed(validate(3), () => 0)
```

firstSuccessOf

The `firstSuccessOf` operator simplifies running a series of effects and returns the result of the first one that succeeds. If none of the effects succeed, the resulting effect will fail with the error of the last effect in the series.

This operator utilizes `Effect.orElse` to combine multiple effects into a single effect.

In the following example, we attempt to retrieve a configuration from different nodes. If retrieving from the primary node fails, we successively try retrieving from the next available nodes until we find a successful result:

```
import { Effect, Console } from "effect"

interface Config {
  // ...
}

const makeConfig = (/* ... */): Config => ({})

const remoteConfig = (name: string): Effect.Effect<Config, Error> =>
  Effect.gen(function* () {
    if (name === "node3") {
      yield* Console.log(`Config for ${name} found`)
      return makeConfig()
    } else {
      yield* Console.log(`Unavailable config for ${name}`)
      return yield* Effect.fail(new Error())
    }
  })

const masterConfig = remoteConfig("master")

const nodeConfigs = ["node1", "node2", "node3", "node4"].map(remoteConfig)

const config = Effect.firstSuccessOf([masterConfig, ...nodeConfigs])

console.log(Effect.runSync(config))
/*
Output:
Unavailable config for master
Unavailable config for node1
Unavailable config for node2
Config for node3 found
{}
*/
```

<Warning> If the collection provided to the `Effect.firstSuccessOf` function is empty, it will throw an `IllegalArgumentException` error. </Warning>

Retrying

Location: 400-guides/150-error-management/600-retrying

Learn how to enhance the resilience of your applications by mastering the retrying capabilities of Effect. Explore `retry`, `retryN`, and `retryOrElse` functions, along with scheduling policies, to automatically handle transient failures. Whether dealing with network requests, database interactions, or other error-prone operations, discover how Effect simplifies the implementation of robust retry strategies.

In software development, it's common to encounter situations where an operation may fail temporarily due to various factors such as network issues, resource unavailability, or external dependencies. In such cases, it's often desirable to retry the operation automatically, allowing it to succeed eventually.

Retrying is a powerful mechanism to handle transient failures and ensure the successful execution of critical operations. In Effect retrying is made simple and flexible with built-in functions and scheduling strategies.

In this guide, we will explore the concept of retrying in Effect and learn how to use the `retry` and `retryOrElse` functions to handle failure scenarios. We'll see how to define retry policies using schedules, which dictate when and how many times the operation should be retried.

Whether you're working on network requests, database interactions, or any other potentially error-prone operations, mastering the retrying capabilities of Effect can significantly enhance the resilience and reliability of your applications.

To demonstrate the functionality of different retry functions, we will be working with the following helper that simulates an effect with possible failures:

```
import { Effect } from "effect"
```

```
let count = 0

// Simulates an effect with possible failures
export const effect = Effect.async<string, Error>((resume) => {
  if (count <= 2) {
    count++
    console.log("failure")
    resume(Effect.fail(new Error()))
  } else {
    console.log("success")
    resume(Effect.succeed("yay!"))
  }
})

// @include: fake
```

retry

The basic syntax of `retry` is as follows:

```
Effect.retry(effect, policy)
```

Example

```
// @filename: fake.ts
// @include: fake

// @filename: index.ts
// ---cut---
import { Effect, Schedule } from "effect"
import { effect } from "./fake"

// Define a repetition policy using a fixed delay between retries
const policy = Schedule.fixed("100 millis")

const repeated = Effect.retry(effect, policy)

Effect.runPromise(repeated).then(console.log)
/*
Output:
failure
failure
failure
success
yay!
*/
```

retry n times

There is a shortcut when the policy is trivial and the failed effect is immediately retried:

```
// @filename: fake.ts
// @include: fake

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import { effect } from "./fake"

Effect.runPromise(Effect.retry(effect, { times: 5 }))
/*
Output:
failure
failure
failure
success
*/
```

retryOrElse

There is another version of `retry` that allows us to define a fallback strategy in case of errors. If something goes wrong, we can handle it using the `retryOrElse` function. It lets us add an `orElse` callback that will run when the repetition fails.

The basic syntax of `retryOrElse` is as follows:

```
Effect.retryOrElse(effect, policy, fallback)
```

Example

```
// @filename: fake.ts
// @include: fake

// @filename: index.ts
// ---cut---
import { Effect, Schedule, Console } from "effect"
```

```
import { effect } from "./fake"

const policy = Schedule.addDelay(
  Schedule.recurs(2), // Retry for a maximum of 2 times
  () => "100 millis" // Add a delay of 100 milliseconds between retries
)

// Create a new effect that retries the effect with the specified policy,
// and provides a fallback effect if all retries fail
const repeated = Effect.retryOrElse(effect, policy, () =>
  Console.log("orElse").pipe(Effect.as("default value"))
)

Effect.runPromise(repeated).then(console.log)
/*
Output:
failure
failure
failure
orElse
default value
*/
```

Two Types of Errors

Location: 400-guides/150-error-management/100-two-error-types

Explore the distinction between expected and unexpected errors in Effect programs. Learn how developers handle anticipated errors, known as "expected errors," and discover the nature of "unexpected errors" that occur outside the normal program flow. Understand how Effect provides detailed failure messages to aid in error recovery.

Just like any other program, Effect programs may fail for expected or unexpected reasons. The difference between a non-Effect program and an Effect program is in the detail provided to you when your program fails. Effect attempts to preserve as much information as possible about what caused your program to fail to produce a detailed, comprehensive, and human readable failure message.

In an Effect program, there are two possible ways for a program to fail:

- **Expected Errors**: These are errors that developers anticipate and expect as part of normal program execution.
- **Unexpected Errors**: These are errors that occur unexpectedly and are not part of the intended program flow.

Expected Errors

These errors, also referred to as *failures*, *typed errors* or *recoverable errors*, are errors that developers anticipate as part of the normal program execution. They serve a similar purpose to checked exceptions and play a role in defining the program's domain and control flow.

Expected errors **are tracked** at the type level by the `Effect` data type in the "Error" channel.

In the following example, it is evident from the type that the program can fail with an error of type `HttpError`:

```
Effect<string, HttpError>
```

Unexpected Errors

Unexpected errors, also referred to as *defects*, *untyped errors*, or *unrecoverable errors*, are errors that developers do not anticipate occurring during normal program execution. Unlike expected errors, which are considered part of a program's domain and control flow, unexpected errors resemble unchecked exceptions and lie outside the expected behavior of the program.

Since these errors are not expected, Effect **does not track** them at the type level. However, the Effect runtime does keep track of these errors and provides several methods to aid in recovering from unexpected errors.

Matching

Location: 400-guides/150-error-management/500-matching

Discover how to handle both success and failure cases in the Effect data type using functions like `match` and `matchEffect`. Learn techniques to perform side effects, ignore values, and access the full cause of failures. Effectively manage control flow and handle errors in your Effect programs.

In the `Effect` data type, just like other data types such as Option and Exit, we have a `match` function that allows us to handle different cases simultaneously. When working with effects, we also have several functions that enable us to handle both success and failure scenarios.

match

The `Effect.match` function allows us to **handle both success and failure cases** in a non-effectful manner by providing a handler for each case:

```
import { Effect } from "effect"

const success: Effect.Effect<number, Error> = Effect.succeed(42)
const failure: Effect.Effect<number, Error> = Effect.fail(new Error("Uh oh!"))

const program1 = Effect.match(success, {
  onFailure: (error) => `failure: ${error.message}`,
  onSuccess: (value) => `success: ${value}`
})

Effect.runPromise(program1).then(console.log) // Output: "success: 42"

const program2 = Effect.match(failure, {
  onFailure: (error) => `failure: ${error.message}`,
  onSuccess: (value) => `success: ${value}`
})

Effect.runPromise(program2).then(console.log) // Output: "failure: Uh oh!"
```

We can also choose to ignore the success and failure values if we're not interested in them:

```
import { Effect } from "effect"
import { constVoid } from "effect/Function"

const task = Effect.fail("Uh oh!").pipe(Effect.as(5))

const program = Effect.match(task, {
  onFailure: constVoid,
  onSuccess: constVoid
})
```

In this case, we use the `constVoid` function from the `Function` module, which constantly returns `void`, to provide handlers that perform no operation. This effectively discards the success and failure values and focuses solely on the control flow or side effects of the program. Alternatively, we can achieve the same result using the `Effect.ignore` function:

```
import { Effect } from "effect"

const task = Effect.fail("Uh oh!").pipe(Effect.as(5))

const program = Effect.ignore(task)
```

matchEffect

In addition to `Effect.match`, we have the `Effect.matchEffect` function, which allows us to handle success and failure cases while performing **additional side effects**. Let's see an example:

```
import { Effect } from "effect"

const success: Effect.Effect<number, Error> = Effect.succeed(42)
const failure: Effect.Effect<number, Error> = Effect.fail(new Error("Uh oh!"))

const program1 = Effect.matchEffect(success, {
  onFailure: (error) =>
    Effect.succeed(`failure: ${error.message}`).pipe(Effect.tap(Effect.log)),
  onSuccess: (value) =>
    Effect.succeed(`success: ${value}`).pipe(Effect.tap(Effect.log))
})

console.log(Effect.runSync(program1))
/*
Output:
... message="success: 42"
success: 42
*/

const program2 = Effect.matchEffect(failure, {
  onFailure: (error) =>
    Effect.succeed(`failure: ${error.message}`).pipe(Effect.tap(Effect.log)),
  onSuccess: (value) =>
    Effect.succeed(`success: ${value}`).pipe(Effect.tap(Effect.log))
})

console.log(Effect.runSync(program2))
/*
Output:
... message="failure: Uh oh!"
failure: Uh oh!
*/
```

In this example, we use `Effect.matchEffect` instead of `Effect.match`. The `Effect.matchEffect` function allows us to perform additional side effects while handling success and failure cases. We can log messages or perform other side effects within the respective handlers.

matchCause / matchCauseEffect

Effect also provides `Effect.matchCause` and `Effect.matchCauseEffect` functions, which are useful for **accessing the full cause** of the underlying fiber in case of failure. This allows us to handle different failure causes separately and take appropriate actions. Here's an example:

```
import { Effect, Console } from "effect"

declare const exceptionalEffect: Effect.Effect<void, Error>

const program = Effect.matchCauseEffect(exceptionalEffect, {
  onFailure: (cause) => {
    switch (cause._tag) {
      case "Fail":
        return Console.log(`Fail: ${cause.error.message}`)
      case "Die":
        return Console.log(`Die: ${cause.defect}`)
      case "Interrupt":
        return Console.log(`${cause.fiberId} interrupted!`)
    }
    return Console.log("failed due to other causes")
  },
  onSuccess: (value) => Console.log(`succeeded with ${value} value`)
})
```

In this example, we have an `exceptionalEffect` that may fail or encounter other types of exceptions. The `matchCauseEffect` function allows us to match and handle different failure causes separately.

Sandboxing

Location: 400-guides/150-error-management/800-sandboxing

Learn how to utilize the powerful `Effect.sandbox` function to encapsulate and understand the causes of errors in your code. This guide explores error sandboxing, allowing you to expose detailed causes of errors, and demonstrates how to handle specific error conditions effectively using standard error-handling operators like `Effect.catchAll` and `Effect.catchTags`.

Errors are a common part of programming, and they can happen for various reasons, such as failures, defects, fiber interruptions, or even a combination of these factors. In this guide, we'll explore how to use the `Effect.sandbox` function to isolate and understand the causes of errors in your code.

sandbox

The `Effect.sandbox` function is a valuable tool that allows you to encapsulate all the potential causes of an error in an effect. It exposes the full cause of an effect, whether it's due to a failure, defect, fiber interruption, or a combination of these factors.

Here's the signature of the `Effect.sandbox` function:

```
sandbox: Effect<A, E, R> -> Effect<A, Cause<E>, R>
```

In simple terms, it takes an effect `Effect<A, E, R>` and transforms it into an effect `Effect<A, Cause<E>, R>` where the error channel now contains a detailed cause of the error.

By using the `Effect.sandbox` function, you gain access to the underlying causes of exceptional effects. These causes are represented as a type of `Cause<E>` and are available in the error channel of the `Effect` data type.

Once you have exposed the causes, you can utilize standard error-handling operators like `Effect.catchAll` and `Effect.catchTags` to handle errors more effectively. These operators allow you to respond to specific error conditions.

Let's walk through an example to illustrate how error sandboxing works:

```
import { Effect, Console } from "effect"

const effect = Effect.fail("Oh uh!").pipe(Effect.as("primary result"))

const sandboxed = Effect.sandbox(effect)

const program = Effect.catchTags(sandboxed, {
  Die: (cause) =>
    Console.log(`Caught a defect: ${cause.defect}`).pipe(
      Effect.as("fallback result on defect")
    ),
  Interrupt: (cause) =>
    Console.log(`Caught a defect: ${cause.fiberId}`).pipe(
      Effect.as("fallback result on fiber interruption")
    ),
  Fail: (cause) =>
    Console.log(`Caught a defect: ${cause.error}`).pipe(
      Effect.as("fallback result on failure")
    )
})

const main = Effect.unsandbox(program)

Effect.runPromise(main).then(console.log)
/*
```

```
Output:
Caught a defect: Oh uh!
fallback result on failure
*/
```

In this example, we expose the full cause of an effect using `Effect.sandbox`. Then, we handle specific error conditions using `Effect.catchTags`. Finally, if needed, we can undo the sandboxing operation with `Effect.unsandbox`.

Error Management

Location: 400-guides/150-error-management/index

Error Management

Timing out

Location: 400-guides/150-error-management/700-timing-out

Learn how to set time constraints on operations with `Effect.timeout`. Discover how to handle scenarios where tasks need to complete within a specified timeframe. Explore variations like `timeoutTo`, `timeoutFail`, and `timeoutFailCause` to customize behavior when a timeout occurs, providing more control and flexibility in managing time-sensitive operations.

In the world of programming, we often deal with tasks that take some time to complete. Sometimes, we want to set a limit on how long we are willing to wait for a task to finish. This is where the `Effect.timeout` function comes into play. It allows us to put a time constraint on an operation, ensuring that it doesn't run indefinitely.

Basic Usage

The `Effect.timeout` function employs a Duration parameter to establish a time limit on an operation. If the operation exceeds this limit, a `TimeoutException` is triggered, indicating a timeout has occurred.

Here's a basic example where `Effect.timeout` is applied to an operation:

```
import { Effect } from "effect"

const myEffect = Effect.gen(function* () {
  console.log("Start processing...")
  yield* Effect.sleep("2 seconds") // Simulates a delay in processing // Simulates a delay in processing
  console.log("Processing complete.")
  return "Result"
})

// wraps this effect, setting a maximum allowable duration of 3 seconds
const timedEffect = myEffect.pipe(Effect.timeout("3 seconds"))

// Output will show that the task completes successfully
// as it falls within the timeout duration
Effect.runPromiseExit(timedEffect).then(console.log)
/*
Output:
Start processing...
Processing complete.
{ _id: 'Exit', _tag: 'Success', value: 'Result' }
*/
```

Handling Timeouts

When an operation does not finish within the specified duration, the behavior of the `Effect.timeout` depends on whether the operation is uninterruptible.

<Info> An **uninterruptible** effect is one that, once started, cannot be stopped mid-execution by the timeout mechanism directly. This could be because the operations within the effect need to run to completion to avoid leaving the system in an inconsistent state. </Info>

1. **Interruptible Operation**: If the operation can be interrupted, it is terminated immediately once the timeout threshold is reached, resulting in a `TimeoutException`.

   ```
   import { Effect } from "effect"

   const myEffect = Effect.gen(function* () {
     console.log("Start processing...")
     yield* Effect.sleep("2 seconds") // Simulates a delay in processing
     console.log("Processing complete.")
     return "Result"
   })

   const timedEffect = myEffect.pipe(Effect.timeout("1 second"))

   Effect.runPromiseExit(timedEffect).then(console.log)
   /*
   ```

```
Output:
Start processing...
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: { _tag: 'TimeoutException' } }
}
*/
```

2. **Uninterruptible Operation**: If the operation is uninterruptible, it continues until completion before the `TimeoutException` is assessed.

```
import { Effect } from "effect"

const myEffect = Effect.gen(function* () {
  console.log("Start processing...")
  yield* Effect.sleep("2 seconds") // Simulates a delay in processing
  console.log("Processing complete.")
  return "Result"
})

const timedEffect = myEffect.pipe(
  Effect.uninterruptible,
  Effect.timeout("1 second")
)

// Outputs a TimeoutException after the task completes, because the task is uninterruptible
Effect.runPromiseExit(timedEffect).then(console.log)
/*
Output:
Start processing...
Processing complete.
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: { _tag: 'TimeoutException' } }
}
*/
```

Disconnection on Timeout

The `Effect.disconnect` function is used to handle timeouts in a nuanced way, particularly when dealing with uninterruptible effects.

It allows the uninterruptible effect to complete its operations in the background, while the main control flow proceeds as if a timeout had occurred.

Here's the distinction:

- **Without Effect.disconnect**:
  - An uninterruptible effect will ignore the timeout and continue executing until it completes, after which the timeout error is assessed.
  - This can lead to delays in recognizing a timeout condition because the system must wait for the effect to complete.
- **With Effect.disconnect**:
  - The uninterruptible effect is allowed to continue in the background, independent of the main control flow.
  - The main control flow recognizes the timeout immediately and proceeds with the timeout error or alternative logic, without having to wait for the effect to complete.
  - This method is particularly useful when the operations of the effect do not need to block the continuation of the program, despite being marked as uninterruptible.

Example

Consider a scenario where a long-running data processing task is initiated, and you want to ensure the system remains responsive, even if the data processing takes too long:

```
import { Effect } from "effect"

const longRunningTask = Effect.gen(function* () {
  console.log("Start heavy processing...")
  yield* Effect.sleep("5 seconds") // Simulate a long process
  console.log("Heavy processing done.")
  return "Data processed"
})

const timedEffect = longRunningTask.pipe(
  Effect.uninterruptible,
  Effect.disconnect, // Allows the task to finish independently if it times out
  Effect.timeout("1 second")
)

Effect.runPromiseExit(timedEffect).then(console.log)
/*
Output:
Start heavy processing...
{
  _id: 'Exit',
```

```
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: { _tag: 'TimeoutException' } }
}
Heavy processing done.
*/
```

Customizing Timeout Behavior

In addition to the basic `Effect.timeout` function, there are variations available that allow you to customize the behavior when a timeout occurs.

timeoutTo

The `timeoutTo` function is similar to `Effect.timeout`, but it provides more control over the final result type. It allows you to define alternative outcomes for both successful and timed-out operations:

```
import { Effect, Either } from "effect"

const myEffect = Effect.gen(function* () {
  console.log("Start processing...")
  yield* Effect.sleep("2 seconds") // Simulates a delay in processing
  console.log("Processing complete.")
  return "Result"
})

const main = myEffect.pipe(
  Effect.timeoutTo({
    duration: "1 second",
    // let's return an Either
    onSuccess: (result): Either.Either<string, string> =>
      Either.right(result),
    onTimeout: (): Either.Either<string, string> => Either.left("Timed out!")
  })
)

Effect.runPromise(main).then(console.log)
/*
Output:
Start processing...
{
  _id: "Either",
  _tag: "Left",
  left: "Timed out!"
}
*/
```

timeoutFail

The `timeoutFail` function allows you to produce a specific error when a timeout happens:

```
import { Effect } from "effect"

const myEffect = Effect.gen(function* () {
  console.log("Start processing...")
  yield* Effect.sleep("2 seconds") // Simulates a delay in processing
  console.log("Processing complete.")
  return "Result"
})

class MyTimeoutError {
  readonly _tag = "MyTimeoutError"
}

const main = myEffect.pipe(
  Effect.timeoutFail({
    duration: "1 second",
    onTimeout: () => new MyTimeoutError()
  })
)

Effect.runPromiseExit(main).then(console.log)
/*
Output:
Start processing...
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: {
    _id: 'Cause',
    _tag: 'Fail',
    failure: MyTimeoutError { _tag: 'MyTimeoutError' }
  }
}
*/
```

timeoutFailCause

The `timeoutFailCause` function allows you to produce a specific defect when a timeout occurs. This is useful when you need to handle timeouts as exceptional cases in your code:

```
import { Effect, Cause } from "effect"

const myEffect = Effect.gen(function* () {
  console.log("Start processing...")
  yield* Effect.sleep("2 seconds") // Simulates a delay in processing
  console.log("Processing complete.")
  return "Result"
})

const main = myEffect.pipe(
  Effect.timeoutFailCause({
    duration: "1 second",
    onTimeout: () => Cause.die("Timed out!")
  })
)

Effect.runPromiseExit(main).then(console.log)
/*
Output:
Start processing...
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Die', defect: 'Timed out!' }
}
*/
```

Parallel and Sequential Errors

Location: 400-guides/150-error-management/1100-sequential-and-parallel-errors

Learn how to handle parallel and sequential errors in Effect programming. Understand the behavior of error handling in scenarios involving parallel computations and sequential operations. Explore combinator `Effect.parallelErrors` to expose and handle multiple parallel failures efficiently.

In a typical Effect application, when an error occurs, it usually fails with the first error encountered by the Effect runtime. Let's look at an example:

```
import { Effect } from "effect"

const fail = Effect.fail("Oh uh!")
const die = Effect.dieMessage("Boom!")

const program = Effect.all([fail, die]).pipe(
  Effect.andThen(die),
  Effect.asVoid
)

Effect.runPromiseExit(program).then(console.log)
/*
Output:
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: 'Oh uh!' }
}
*/
```

In this case, the `program` will fail with the first error, which is "Oh uh!":

Parallel Errors

However, in some situations, you may encounter multiple errors, especially when performing parallel computations. When parallel computations are involved, the application may fail due to multiple errors. Here's an example:

```
import { Effect } from "effect"

const fail = Effect.fail("Oh uh!")
const die = Effect.dieMessage("Boom!")

const program = Effect.all([fail, die], { concurrency: "unbounded" }).pipe(
  Effect.asVoid
)

Effect.runPromiseExit(program).then(console.log)
/*
Output:
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: {
    _id: 'Cause',
    _tag: 'Parallel',
```

```
    left: { _id: 'Cause', _tag: 'Fail', failure: 'Oh uh!' },
    right: { _id: 'Cause', _tag: 'Die', defect: [Object] }
  }
}
*/
```

In this example, the `program` runs both `fail` and `die` concurrently, and if both fail, it will result in multiple errors.

parallelErrors

Effect provides a useful combinator called `Effect.parallelErrors` that exposes all parallel failure errors in the error channel. Here's how you can use it:

```
import { Effect } from "effect"

const fail1 = Effect.fail("Oh uh!")
const fail2 = Effect.fail("Oh no!")
const die = Effect.dieMessage("Boom!")

const program = Effect.all([fail1, fail2, die], {
  concurrency: "unbounded"
}).pipe(Effect.asVoid, Effect.parallelErrors)

Effect.runPromiseExit(program).then(console.log)
/*
Output:
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: [ 'Oh uh!', 'Oh no!' ] }
}
*/
```

In this example, `Effect.parallelErrors` combines the errors from `fail1` and `fail2` into a single error.

<Warning> Note that this operator is **only for failures**, not defects or interruptions. </Warning>

Sequential Errors

When working with resource-safety operators like `Effect.ensuring`, you may encounter multiple sequential errors. This happens because regardless of whether the original effect has any errors or not, the finalizer is uninterruptible and will run. Here's an example:

```
import { Effect } from "effect"

const fail = Effect.fail("Oh uh!")
const die = Effect.dieMessage("Boom!")

const program = fail.pipe(Effect.ensuring(die))

Effect.runPromiseExit(program).then(console.log)
/*
Output:
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: {
    _id: 'Cause',
    _tag: 'Sequential',
    left: { _id: 'Cause', _tag: 'Fail', failure: 'Oh uh!' },
    right: { _id: 'Cause', _tag: 'Die', defect: [Object] }
  }
}
*/
```

In this case, the `program` will result in multiple sequential errors if both `fail` and the finalizer `die` encounter errors.

Error Accumulation

Location: 400-guides/150-error-management/900-error-accumulation

Discover how to handle errors in your Effect programming by exploring sequential combinators, such as `Effect.zip` and `Effect.forEach`. Learn about the "fail fast" policy and explore alternative approaches for error accumulation using functions like `Effect.validate`, `Effect.validateAll`, `Effect.validateFirst`, and `Effect.partition`.

Sequential combinators such as `Effect.zip` and `Effect.forEach` have a "fail fast" policy when it comes to error management. This means that they stop and return immediately when they encounter the first error.

Let's take a look at an example using the `Effect.zip` operator. In this example, we can see that `Effect.zip` will fail as soon as it encounters the first failure. As a result, only the first error is displayed:

```
import { Effect } from "effect"
```

```
const task1 = Effect.succeed(1)
const task2 = Effect.fail("Oh uh!").pipe(Effect.as(2))
const task3 = Effect.succeed(3)
const task4 = Effect.fail("Oh no!").pipe(Effect.as(4))

const program = task1.pipe(
  Effect.zip(task2),
  Effect.zip(task3),
  Effect.zip(task4)
)

Effect.runPromise(program).then(console.log, console.error)
/*
Output:
(FiberFailure) Error: Oh uh!
*/
```

The `Effect.forEach` function behaves similarly. It takes a collection and an effectful operation, and tries to apply the operation to all elements of the collection. However, it also follows the "fail fast" policy and fails when it encounters the first error:

```
import { Effect } from "effect"

const program = Effect.forEach([1, 2, 3, 4, 5], (n) => {
  if (n < 4) {
    return Effect.succeed(n)
  } else {
    return Effect.fail(`${n} is not less that 4`)
  }
})

Effect.runPromise(program).then(console.log, console.error)
/*
Output:
(FiberFailure) Error: 4 is not less that 4
*/
```

However, there are situations where we may need to collect all potential errors in a computation instead of failing fast. In such cases, we can use operators that accumulate errors as well as successes.

validate

The `Effect.validate` function is similar to `Effect.zip`, but if it encounters an error, it continues the zip operation instead of stopping. It combines the effects and accumulates both errors and successes:

```
import { Effect } from "effect"

const task1 = Effect.succeed(1)
const task2 = Effect.fail("Oh uh!").pipe(Effect.as(2))
const task3 = Effect.succeed(3)
const task4 = Effect.fail("Oh no!").pipe(Effect.as(4))

const program = task1.pipe(
  Effect.validate(task2),
  Effect.validate(task3),
  Effect.validate(task4)
)

Effect.runPromise(program).then(console.log, console.error)
/*
Output:
(FiberFailure) Error: Oh uh!
Error: Oh no!
*/
```

With `Effect.validate`, we can collect all the errors encountered during the computation instead of stopping at the first error. This allows us to have a complete picture of all the potential errors and successes in our program.

validateAll

The `Effect.validateAll` function is similar to the `Effect.forEach` function. It transforms all elements of a collection using the provided effectful operation, but it collects all errors in the error channel, as well as the success values in the success channel.

```
import { Effect } from "effect"

const program = Effect.validateAll([1, 2, 3, 4, 5], (n) => {
  if (n < 4) {
    return Effect.succeed(n)
  } else {
    return Effect.fail(`${n} is not less that 4`)
  }
})

Effect.runPromise(program).then(console.log, console.error)
/*
Output:
```

```
(FiberFailure) Error: ["4 is not less that 4","5 is not less that 4"]
*/
```

<Warning> Note that this function is lossy, which means that if there are any errors, all successes will be lost. If you need a function that preserves both successes and failures, please refer to the partition function. </Warning>

validateFirst

The `Effect.validateFirst` function is similar to `Effect.validateAll` but it will return the first success (or all the failures):

```
import { Effect } from "effect"

const program = Effect.validateFirst([1, 2, 3, 4, 5], (n) => {
  if (n < 4) {
    return Effect.fail(`${n} is not less that 4`)
  } else {
    return Effect.succeed(n)
  }
})

Effect.runPromise(program).then(console.log, console.error)
// Output: 4
```

Please note that the return type is `number` instead of `number[]`, as in the case of `validateAll`.

partition

The `Effect.partition` function takes an iterable and an effectful function that transforms each value of the iterable. It then creates a tuple of both failures and successes in the success channel:

```
import { Effect } from "effect"

const program = Effect.partition([0, 1, 2, 3, 4], (n) => {
  if (n % 2 === 0) {
    return Effect.succeed(n)
  } else {
    return Effect.fail(`${n} is not even`)
  }
})

Effect.runPromise(program).then(console.log, console.error)
// Output: [ [ '1 is not even', '3 is not even' ], [ 0, 2, 4 ] ]
```

Please note that this operator is an unexceptional effect, which means that the type of the error channel is `never`. Therefore, if we encounter a failure case, the whole effect doesn't fail.

Error Channel Operations

Location: 400-guides/150-error-management/1000-error-channel-operations

Explore various operations on the error channel in Effect, including error mapping, both channel mapping, filtering success values, inspecting errors, exposing errors, merging error and success channels, and flipping error and success channels. Learn how to handle errors effectively in your Effect programming.

In Effect you can perform various operations on the error channel of effects. These operations allow you to transform, inspect, and handle errors in different ways. Let's explore some of these operations.

Map Operations

mapError

The `Effect.mapError` function is used when you need to **transform or modify an error** produced by an effect, without affecting the success value. This can be helpful when you want to add extra information to the error or change its type.

```
import { Effect } from "effect"

const simulatedTask = Effect.fail("Oh no!").pipe(Effect.as(1))

const mapped = Effect.mapError(simulatedTask, (message) => new Error(message))
```

We can observe that the type in the error channel of our program has changed from `string` to `Error`.

<Info> It's important to note that using the `Effect.mapError` function **does not change** the overall success or failure of the effect. If the mapped effect is successful, then the mapping function is ignored. In other words, the `Effect.mapError` operation only applies the transformation to the error channel of the effect, while leaving the success channel unchanged. </Info>

mapBoth

The `Effect.mapBoth` function allows you to **apply transformations to both channels**: the error channel and the success channel of an effect. It takes two map functions as arguments: one for the error channel and the other for the success channel.

```
import { Effect } from "effect"

const simulatedTask = Effect.fail("Oh no!").pipe(Effect.as(1))

const modified = Effect.mapBoth(simulatedTask, {
  onFailure: (message) => new Error(message),
  onSuccess: (n) => n > 0
})
```

After using `mapBoth`, we can observe that the type of our program has changed from `Effect<number, string>` to `Effect<boolean, Error>`.

<Info> It's important to note that using the `mapBoth` function **does not change** the overall success or failure of the effect. It only transforms the values in the error and success channels while preserving the effect's original success or failure status. </Info>

Filtering the Success Channel

The Effect library provides several operators to **filter values on the success channel** based on a given predicate. These operators offer different strategies for handling cases where the predicate fails. Let's explore them:

Function	Description
`Effect.filterOrFail`	This operator filters the values on the success channel based on a predicate. If the predicate fails for any value, the original effect fails with an error.
`Effect.filterOrDie` and `Effect.filterOrDieMessage`	These operators also filter the values on the success channel based on a predicate. If the predicate fails for any value, the original effect terminates abruptly. The `filterOrDieMessage` variant allows you to provide a custom error message.
`Effect.filterOrElse`	This operator filters the values on the success channel based on a predicate. If the predicate fails for any value, an alternative effect is executed instead.

Here's an example that demonstrates these filtering operators in action:

```
import { Effect, Random, Cause } from "effect"

const task1 = Effect.filterOrFail(
  Random.nextRange(-1, 1),
  (n) => n >= 0,
  () => "random number is negative"
)

const task2 = Effect.filterOrDie(
  Random.nextRange(-1, 1),
  (n) => n >= 0,
  () => new Cause.IllegalArgumentException("random number is negative")
)

const task3 = Effect.filterOrDieMessage(
  Random.nextRange(-1, 1),
  (n) => n >= 0,
  "random number is negative"
)

const task4 = Effect.filterOrElse(
  Random.nextRange(-1, 1),
  (n) => n >= 0,
  () => task3
)
```

It's important to note that depending on the specific filtering operator used, the effect can either fail, terminate abruptly, or execute an alternative effect when the predicate fails. Choose the appropriate operator based on your desired error handling strategy and program logic.

In addition to the filtering capabilities discussed earlier, you have the option to further refine and narrow down the type of the success channel by providing a user-defined type guard to the `filterOr*` APIs. This not only enhances type safety but also improves code clarity. Let's explore this concept through an example:

```
import { Effect, pipe } from "effect"

// Define a user interface
interface User {
  readonly name: string
}

// Assume an asynchronous authentication function
declare const auth: () => Promise<User | null>

const program = pipe(
  Effect.promise(() => auth()),
  Effect.filterOrFail(
    // Define a guard to narrow down the type
    (user): user is User => user !== null,
    () => new Error("Unauthorized")
  ),
```

```
  Effect.andThen((user) => user.name) // The 'user' here has type `User`, not `User | null`
)
```

In the example above, a guard is used within the `filterOrFail` API to ensure that the `user` is of type `User` rather than `User | null`. This refined type information improves the reliability of your code and makes it more understandable.

If you prefer, you can utilize a pre-made guard like Predicate.isNotNull for simplicity and consistency.

Inspecting Errors

Similar to tapping for success values, Effect provides several operators for **inspecting error values**. These operators allow us to peek into failures or underlying defects or causes:

- `tapError`
- `tapBoth`
- `tapErrorCause`
- `tapDefect`

Let's see an example of how to use these operators:

```
import { Effect, Random, Console } from "effect"

const task = Effect.filterOrFail(
  Random.nextRange(-1, 1),
  (n) => n >= 0,
  () => "random number is negative"
)

const tapping1 = Effect.tapError(task, (error) =>
  Console.log(`failure: ${error}`)
)

const tapping2 = Effect.tapBoth(task, {
  onFailure: (error) => Console.log(`failure: ${error}`),
  onSuccess: (randomNumber) => Console.log(`random number: ${randomNumber}`)
})
```

<Info> It's important to note that tapping into error values **does not change** the type of the program. </Info>

Exposing Errors in The Success Channel

You can use the `Effect.either` function to convert an `Effect<A, E, R>` into another effect where both its failure (`E`) and success (`A`) channels have been lifted into an Either<A, E> data type:

```
Effect<A, E, R> -> Effect<Either<A, E>, never, R>
```

The resulting effect is an unexceptional effect, which means it cannot fail, because the failure case has been exposed as part of the `Either` left case. Therefore, the error parameter of the returned Effect is `never`, as it is guaranteed that the effect does not model failure.

This function becomes especially useful when recovering from effects that may fail when using `Effect.gen`.

Exposing the Cause in The Success Channel

You can use the `Effect.cause` function to **expose the cause** of an effect, which is a more detailed representation of failures, including error messages and defects.

Merging the Error Channel into the Success Channel

Using the `Effect.merge` function, you can **merge the error channel into the success channel**, creating an effect that always succeeds with the merged value.

```
import { Effect } from "effect"

const simulatedTask = Effect.fail("Oh uh!").pipe(Effect.as(2))

const merged = Effect.merge(simulatedTask)
```

Flipping Error and Success Channels

Using the `Effect.flip` function, you can **flip the error and success channels** of an effect, effectively swapping their roles.

```
import { Effect } from "effect"

const simulatedTask = Effect.fail("Oh uh!").pipe(Effect.as(2))

const flipped = Effect.flip(simulatedTask)
```

Yieldable Errors

Location: 400-guides/150-error-management/1200-yieldable-errors

Learn about "Yieldable Errors" in Effect programming, a convenient way to handle custom errors within generator functions. Explore the `Data.Error` and `Data.TaggedError` constructors for creating base and tagged yieldable errors, simplifying error handling in your code.

"Yieldable Errors" are special types of errors that can be yielded within a generator function used by `Effect.gen`. The unique feature of these errors is that you don't need to use the `Effect.fail` API explicitly to handle them. They offer a more intuitive and convenient way to work with custom errors in your code.

Data.Error

The `Data.Error` constructor enables you to create a base yieldable error class. This class can be used to represent different types of errors in your code. Here's how you can use it:

```
import { Effect, Data } from "effect"

class MyError extends Data.Error<{ message: string }> {}

export const program = Effect.gen(function* () {
  yield* new MyError({ message: "Oh no!" }) // same as yield* Effect.fail(new MyError({ message: "Oh no!" })
})

Effect.runPromiseExit(program).then(console.log)
/*
Output:
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: { message: 'Oh no!' } }
}
*/
```

Data.TaggedError

The `Data.TaggedError` constructor is useful for creating tagged yieldable errors. These errors bear a distinct property named `_tag`, which acts as their unique identifier, allowing you to differentiate them from one another. Here's how you can use it:

```
import { Effect, Data, Random } from "effect"

// An error with _tag: "Foo"
class FooError extends Data.TaggedError("Foo")<{
  message: string
}> {}

// An error with _tag: "Bar"
class BarError extends Data.TaggedError("Bar")<{
  randomNumber: number
}> {}

export const program = Effect.gen(function* () {
  const n = yield* Random.next
  return n > 0.5
    ? "yay!"
    : n < 0.2
      ? yield* new FooError({ message: "Oh no!" })
      : yield* new BarError({ randomNumber: n })
}).pipe(
  Effect.catchTags({
    Foo: (error) => Effect.succeed(`Foo error: ${error.message}`),
    Bar: (error) => Effect.succeed(`Bar error: ${error.randomNumber}`)
  })
)

Effect.runPromise(program).then(console.log, console.error)
/*
Example Output (n < 0.2):
Foo error: Oh no!
*/
```

In this example, we create `FooError` and `BarError` classes with distinct tags ("Foo" and "Bar"). These tags help identify the type of error when handling errors in your code.

Unexpected Errors

Location: 400-guides/150-error-management/300-unexpected-errors

Learn how Effect handles unrecoverable errors, such as defects, providing functions like `die`, `dieMessage`, `orDie`, and `orDieWith`. Explore techniques to terminate effect execution, handle unexpected errors, and recover from defects. Discover the use of `catchAllDefect` and

`catchSomeDefect` to manage and selectively recover from specific defects.

There are situations where you may encounter unexpected errors, and you need to decide how to handle them. Effect provides functions to help you deal with such scenarios, allowing you to take appropriate actions when errors occur during the execution of your effects.

Creating Unrecoverable Errors

In the same way it is possible to leverage combinators such as `fail` to create values of type `Effect<never, E, never>` the Effect library provides tools to create defects.

Creating defects is a common necessity when dealing with errors from which it is not possible to recover from a business logic perspective, such as attempting to establish a connection that is refused after multiple retries.

In those cases terminating the execution of the effect and moving into reporting, through an output such as stdout or some external monitoring service, might be the best solution.

The following functions and combinators allow for termination of the effect and are often used to convert values of type `Effect<A, E, R>` into values of type `Effect<A, never, R>` allowing the programmer an escape hatch from having to handle and recover from errors for which there is no sensible way to recover.

die / dieMessage

The `Effect.die` function returns an effect that throws a specified error, while `Effect.dieMessage` throws a `RuntimeException` with a specified text message. These functions are useful for terminating a fiber when a defect, a critical and unexpected error, is detected in the code.

Example using `die`:

```
import { Effect } from "effect"

const divide = (a: number, b: number): Effect.Effect<number> =>
  b === 0
    ? Effect.die(new Error("Cannot divide by zero"))
    : Effect.succeed(a / b)

Effect.runSync(divide(1, 0)) // throws Error: Cannot divide by zero
```

Example using `dieMessage`:

```
import { Effect } from "effect"

const divide = (a: number, b: number): Effect.Effect<number> =>
  b === 0 ? Effect.dieMessage("Cannot divide by zero") : Effect.succeed(a / b)

Effect.runSync(divide(1, 0)) // throws RuntimeException: Cannot divide by zero
```

orDie

The `Effect.orDie` function transforms an effect's failure into a termination of the fiber, making all failures unchecked and not part of the type of the effect. It can be used to handle errors that you do not wish to recover from.

```
import { Effect } from "effect"

const divide = (a: number, b: number): Effect.Effect<number, Error> =>
  b === 0
    ? Effect.fail(new Error("Cannot divide by zero"))
    : Effect.succeed(a / b)

const program = Effect.orDie(divide(1, 0))

Effect.runSync(program) // throws Error: Cannot divide by zero
```

After using `Effect.orDie`, the error channel type of the `program` is `never`, indicating that all failures are unchecked, and the effect is expected to terminate the fiber when an error occurs.

orDieWith

Similar to `Effect.orDie`, the `Effect.orDieWith` function transforms an effect's failure into a termination of the fiber using a specified mapping function. It allows you to customize the error message before terminating the fiber.

```
import { Effect } from "effect"

const divide = (a: number, b: number): Effect.Effect<number, Error> =>
  b === 0
    ? Effect.fail(new Error("Cannot divide by zero"))
    : Effect.succeed(a / b)

const program = Effect.orDieWith(
  divide(1, 0),
  (error) => new Error(`defect: ${error.message}`)
)
```

```
Effect.runSync(program) // throws Error: defect: Cannot divide by zero
```

After using `Effect.orDieWith`, the error channel type of the `program` is `never`, just like with `Effect.orDie`.

Catching

Effect provides two functions that allow you to handle unexpected errors that may occur during the execution of your effects.

<Warning> There is no sensible way to recover from defects. The functions we're about to discuss should be used only at the boundary between Effect and an external system, to transmit information on a defect for diagnostic or explanatory purposes. </Warning>

catchAllDefect

The `Effect.catchAllDefect` function allows you to recover from all defects using a provided function. Here's an example:

```
import { Effect, Cause, Console } from "effect"

const program = Effect.catchAllDefect(
  Effect.dieMessage("Boom!"), // Simulating a runtime error
  (defect) => {
    if (Cause.isRuntimeException(defect)) {
      return Console.log(`RuntimeException defect caught: ${defect.message}`)
    }
    return Console.log("Unknown defect caught.")
  }
)

// We get an Exit.Success because we caught all defects
Effect.runPromiseExit(program).then(console.log)
/*
Output:
RuntimeException defect caught: Boom!
{
  _id: "Exit",
  _tag: "Success",
  value: undefined
}
*/
```

It's important to understand that `catchAllDefect` can only handle defects, not expected errors (such as those caused by `Effect.fail`) or interruptions in execution (such as when using `Effect.interrupt`).

A defect refers to an error that cannot be anticipated in advance, and there is no reliable way to respond to it. As a general rule, it's recommended to let defects crash the application, as they often indicate serious issues that need to be addressed.

However, in some specific cases, such as when dealing with dynamically loaded plugins, a controlled recovery approach might be necessary. For example, if our application supports runtime loading of plugins and a defect occurs within a plugin, we may choose to log the defect and then reload only the affected plugin instead of crashing the entire application. This allows for a more resilient and uninterrupted operation of the application.

catchSomeDefect

The `Effect.catchSomeDefect` function in Effect allows you to recover from specific defects using a provided partial function. Let's take a look at an example:

```
import { Effect, Cause, Option, Console } from "effect"

const program = Effect.catchSomeDefect(
  Effect.dieMessage("Boom!"), // Simulating a runtime error
  (defect) => {
    if (Cause.isIllegalArgumentException(defect)) {
      return Option.some(
        Console.log(
          `Caught an IllegalArgumentException defect: ${defect.message}`
        )
      )
    }
    return Option.none()
  }
)

// Since we are only catching IllegalArgumentException
// we will get an Exit.Failure because we simulated a runtime error.
Effect.runPromiseExit(program).then(console.log)
/*
Output:
{
  _id: "Exit",
  _tag: "Failure",
  cause: {
    _id: "Cause",
    _tag: "Die",
    defect: {
      _tag: "RuntimeException",
```

```
      message: "Boom!",
      [Symbol(@effect/io/Cause/errors/RuntimeException)]: Symbol(@effect/io/Cause/errors/RuntimeException),
      toString: [Function: toString]
    }
  }
}
*/
```

It's important to understand that `catchSomeDefect` can only handle defects, not expected errors (such as those caused by `Effect.fail`) or interruptions in execution (such as when using `Effect.interrupt`).

Expected Errors

Location: 400-guides/150-error-management/200-expected-errors

Explore how Effect represents and manages expected errors. Learn about creating error instances, tracking errors at the type level, and the short-circuiting behavior of Effect programs. Discover techniques to catch and recover from errors, and gain insights into error handling strategies using Effect's powerful combinators.

In this guide you will learn:

- How Effect represents expected errors
- The tools Effect provides for robust and comprehensive error management

As we saw in the guide Creating Effects, we can use the `fail` constructor to create an Effect that represents an error:

```
import { Effect } from "effect"

class HttpError {
  readonly _tag = "HttpError"
}

const program = Effect.fail(new HttpError())
```

<Info> We use a class to represent the `HttpError` type above simply to gain access to both the error type and a free constructor. However, you can use whatever you like to model your error types. </Info>

It's worth noting that we added a readonly `_tag` field as discriminant to our error in the example:

```
class HttpError {
  readonly _tag = "HttpError"
}
```

<Idea> Adding a discriminant field, such as `_tag`, can be beneficial for distinguishing between different types of errors during error handling. It also prevents TypeScript from unifying types, ensuring that each error is treated uniquely based on its discriminant value. </Idea>

Expected errors **are tracked at the type level** by the `Effect` data type in the "Error" channel.

It is evident from the type of `program` that can fail with an error of type `HttpError`:

```
Effect<never, HttpError, never>
```

Error Tracking

The following program serves as an illustration of how errors are automatically tracked for you:

Effect automatically keeps track of the possible errors that can occur during the execution of the program. In this case, we have `FooError` and `BarError` as the possible error types. The error channel of the `program` is specified as

```
Effect<string, FooError | BarError, never>
```

indicating that it can potentially fail with either a `FooError` or a `BarError`.

Short-Circuiting

When working with APIs like `Effect.gen`, `Effect.map`, `Effect.flatMap`, `Effect.andThen` and `Effect.all`, it's important to understand how they handle errors. These APIs are designed to **short-circuit the execution** upon encountering the **first error**.

What does this mean for you as a developer? Well, let's say you have a chain of operations or a collection of effects to be executed in sequence. If any error occurs during the execution of one of these effects, the remaining computations will be skipped, and the error will be propagated to the final result.

In simpler terms, the short-circuiting behavior ensures that if something goes wrong at any step of your program, it won't waste time executing unnecessary computations. Instead, it will immediately stop and return the error to let you know that something went wrong.

This code snippet demonstrates the short-circuiting behavior when an error occurs. Each operation depends on the successful execution of the previous one. If any error occurs, the execution is short-circuited, and the error is propagated. In this specific example, `task3` is never executed because an error occurs in `task2`.

Catching all Errors

either

The `Effect.either` function converts an `Effect<A, E, R>` into another effect where both its failure (`E`) and success (`A`) channels have been lifted into an Either<A, E> data type:

```
Effect<A, E, R> -> Effect<Either<A, E>, never, R>
```

The Either<R, L> data type represents a value that can be either a `Right` value (`R`) or a `Left` value (`L`). By yielding an `Either`, we gain the ability to "pattern match" on this type to handle both failure and success cases within the generator function.

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect, Either } from "effect"
import { program } from "./error-tracking"

const recovered = Effect.gen(function* () {
  const failureOrSuccess = yield* Effect.either(program)
  if (Either.isLeft(failureOrSuccess)) {
    // failure case: you can extract the error from the `left` property
    const error = failureOrSuccess.left
    return `Recovering from ${error._tag}`
  } else {
    // success case: you can extract the value from the `right` property
    return failureOrSuccess.right
  }
})
```

We can make the code less verbose by using the `Either.match` function, which directly accepts the two callback functions for handling errors and successful values:

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect, Either } from "effect"
import { program } from "./error-tracking"

const recovered = Effect.gen(function* () {
  const failureOrSuccess = yield* Effect.either(program)
  return Either.match(failureOrSuccess, {
    onLeft: (error) => `Recovering from ${error._tag}`,
    onRight: (value) => value // do nothing in case of success
  })
})
```

catchAll

The `Effect.catchAll` function allows you to catch any error that occurs in the program and provide a fallback.

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import { program } from "./error-tracking"

const recovered = program.pipe(
  Effect.catchAll((error) => Effect.succeed(`Recovering from ${error._tag}`))
)
```

We can observe that the type in the error channel of our program has changed to `never`, indicating that all errors have been handled.

Catching Some Errors

Suppose we want to handle a specific error, such as `FooError`.

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect, Either } from "effect"
import { program } from "./error-tracking"

const recovered = Effect.gen(function* () {
  const failureOrSuccess = yield* Effect.either(program)
  if (Either.isLeft(failureOrSuccess)) {
```

```
    const error = failureOrSuccess.left
    if (error._tag === "FooError") {
      return "Recovering from FooError"
    }
    return yield* Effect.fail(error)
  } else {
    return failureOrSuccess.right
  }
})
```

We can observe that the type in the error channel of our program has changed to only show `BarError`, indicating that `FooError` has been handled.

If we also want to handle `BarError`, we can easily add another case to our code:

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect, Either } from "effect"
import { program } from "./error-tracking"

const recovered = Effect.gen(function* () {
  const failureOrSuccess = yield* Effect.either(program)
  if (Either.isLeft(failureOrSuccess)) {
    const error = failureOrSuccess.left
    if (error._tag === "FooError") {
      return "Recovering from FooError"
    } else {
      return "Recovering from BarError"
    }
  } else {
    return failureOrSuccess.right
  }
})
```

We can observe that the type in the error channel of our program has changed to `never`, indicating that all errors have been handled.

catchSome

If we want to catch and recover from only some types of errors and effectfully attempt recovery, we can use the `Effect.catchSome` function:

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect, Option } from "effect"
import { program } from "./error-tracking"

const recovered = program.pipe(
  Effect.catchSome((error) => {
    if (error._tag === "FooError") {
      return Option.some(Effect.succeed("Recovering from FooError"))
    }
    return Option.none()
  })
)
```

In the code above, `Effect.catchSome` takes a function that examines the error (`error`) and decides whether to attempt recovery or not. If the error matches a specific condition, recovery can be attempted by returning `Option.some(effect)`. If no recovery is possible, you can simply return `Option.none()`.

It's important to note that while `Effect.catchSome` lets you catch specific errors, it **doesn't alter the error type** itself. Therefore, the resulting effect (`recovered` in this case) will still have the same error type (`FooError | BarError`) as the original effect.

catchIf

Similar to `Effect.catchSome`, the function `Effect.catchIf` allows you to recover from specific errors based on a predicate:

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import { program } from "./error-tracking"

const recovered = program.pipe(
  Effect.catchIf(
    (error) => error._tag === "FooError",
    () => Effect.succeed("Recovering from FooError")
  )
)
```

It's important to note that for TypeScript versions < 5.5, while `Effect.catchIf` lets you catch specific errors, it **doesn't alter the error type** itself. Therefore, the resulting effect (`recovered` in this case) will still have the same error type (`FooError | BarError`) as the original effect. In TypeScript versions >= 5.5, improved type narrowing causes the resulting error type to be inferred as `BarError`.

For TypeScript versions < 5.5, if you provide a user-defined type guard instead of a predicate, the resulting error type will be pruned, returning an `Effect<string, BarError, never>`:

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import { program, FooError } from "./error-tracking"

const recovered = program.pipe(
  Effect.catchIf(
    (error): error is FooError => error._tag === "FooError",
    () => Effect.succeed("Recovering from FooError")
  )
)
```

catchTag

If your program's errors are all tagged with a `_tag` field that acts as a discriminator you can use the `Effect.catchTag` function to catch and handle specific errors with precision.

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import { program } from "./error-tracking"

const recovered = program.pipe(
  Effect.catchTag("FooError", (_fooError) =>
    Effect.succeed("Recovering from FooError")
  )
)
```

In the example above, the `Effect.catchTag` function allows us to handle `FooError` specifically. If a `FooError` occurs during the execution of the program, the provided error handler function will be invoked, and the program will proceed with the recovery logic specified within the handler.

We can observe that the type in the error channel of our program has changed to only show `BarError`, indicating that `FooError` has been handled.

If we also wanted to handle `BarError`, we can simply add another `catchTag`:

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import { program } from "./error-tracking"

const recovered = program.pipe(
  Effect.catchTag("FooError", (_fooError) =>
    Effect.succeed("Recovering from FooError")
  ),
  Effect.catchTag("BarError", (_barError) =>
    Effect.succeed("Recovering from BarError")
  )
)
```

We can observe that the type in the error channel of our program has changed to `never`, indicating that all errors have been handled.

<Warning> It is important to ensure that the error type used with `catchTag` has a `readonly _tag` discriminant field. This field is required for the matching and handling of specific error tags. </Warning>

catchTags

Instead of using the `Effect.catchTag` function multiple times to handle individual error types, we have a more convenient option called `Effect.catchTags`. With `Effect.catchTags`, we can handle multiple errors in a single block of code.

```
// @filename: error-tracking.ts
// @include: error-tracking

// @filename: index.ts
// ---cut---
import { Effect } from "effect"
import { program } from "./error-tracking"

const recovered = program.pipe(
  Effect.catchTags({
```

```
    FooError: (_fooError) => Effect.succeed(`Recovering from FooError`),
    BarError: (_barError) => Effect.succeed(`Recovering from BarError`)
  })
)
```

In the above example, instead of using `Effect.catchTag` multiple times to handle individual errors, we utilize the `Effect.catchTags` combinator. This combinator takes an object where each property represents a specific error `_tag` (`"FooError"` and `"BarError"` in this case), and the corresponding value is the error handler function to be executed when that particular error occurs.

<Warning> It is important to ensure that all the error types used with `Effect.catchTags` have a `readonly _tag` discriminant field. This field is required for the matching and handling of specific error tags. </Warning>

Cache

Location: 400-guides/630-cache

Cache

The Cache module makes it easy to optimize the performance of our application by caching values.

Introduction

In many applications, we may encounter scenarios where overlapping work is performed. For example, if we are developing a service that handles incoming requests, it is essential to avoid processing duplicate requests. By using the Cache module, we can enhance our application's performance by preventing redundant work.

Key Features of Cache:

- **Compositionality**: Cache allows different parts of our application to perform overlapping work while still benefiting from compositional programming principles.
- **Unification of Synchronous and Asynchronous Caches**: The compositional definition of a cache through a lookup function unifies both synchronous and asynchronous caches, allowing the lookup function to compute values either synchronously or asynchronously.
- **Deep Effect Integration**: Cache is designed to work natively with the Effect library, supporting concurrent lookups, failure handling, and interruption without losing the power of Effect.
- **Caching Policy**: Caching policies determine when values should be removed from the cache, providing flexibility for complex and custom caching strategies. The policy has two parts:
  - **Priority (Optional Removal)**: Defines the order in which values **might** be removed when the cache is running out of space.
  - **Evict (Mandatory Removal)**: Specifies when values **must** be removed because they are no longer valid (e.g., they are too old or no longer satisfy business requirements).
- **Composition Caching Policy**: Allows the definition of complex caching policies using simpler ones.
- **Cache/Entry Statistics**: Cache tracks metrics such as entries, hits, misses, helping us to assess and optimize cache performance.

How to Define a Cache?

A cache is defined by a lookup function that describes how to compute the value associated with a key if it is not already in the cache.

```
export type Lookup<Key, Value, Error = never, Environment = never> = (
  key: Key
) => Effect.Effect<Value, Error, Environment>
```

The lookup function takes a key of type `Key` and returns an `Effect` that requires an environment of type `Environment` and can fail with an error of type `Error` or succeed with a value of type `Value`. Since the lookup function returns an `Effect`, it can describe both synchronous and asynchronous workflows.

In short, if you can describe it with an `Effect`, you can use it as the lookup function for a cache.

We construct a cache using a lookup function along with a maximum size and a time to live.

```
export declare const make: <
  Key,
  Value,
  Error = never,
  Environment = never
>(options: {
  readonly capacity: number
  readonly timeToLive: Duration.DurationInput
  readonly lookup: Lookup<Key, Value, Error, Environment>
}) => Effect.Effect<Cache<Key, Value, Error>, never, Environment>
```

Once a cache is created, the most idiomatic way to work with it is the `get` operator. The `get` operator returns the current value in the cache if it exists, or computes a new value, puts it in the cache, and returns it.

If multiple concurrent processes request the same value, it will only be computed once. All other processes will receive the computed value as soon as it is available. This is managed using Effect's fiber-based concurrency model without blocking the underlying thread.

Example

In this example, we call `timeConsumingEffect` three times in parallel with the same key. The Cache runs this effect only once, so concurrent lookups will wait until the value is available:

```
import { Effect, Cache, Duration } from "effect"

const timeConsumingEffect = (key: string) =>
  Effect.sleep("2 seconds").pipe(Effect.as(key.length))

const program = Effect.gen(function* () {
  const cache = yield* Cache.make({
    capacity: 100,
    timeToLive: Duration.infinity,
    lookup: timeConsumingEffect
  })
  const result = yield* cache
    .get("key1")
    .pipe(
      Effect.zip(cache.get("key1"), { concurrent: true }),
      Effect.zip(cache.get("key1"), { concurrent: true })
    )
  console.log(
    `Result of parallel execution of three effects with the same key: ${result}`
  )

  const hits = yield* cache.cacheStats.pipe(Effect.map((_) => _.hits))
  const misses = yield* cache.cacheStats.pipe(Effect.map((_) => _.misses))
  console.log(`Number of cache hits: ${hits}`)
  console.log(`Number of cache misses: ${misses}`)
})

Effect.runPromise(program)
/*
Output:
Result of parallel execution of three effects with the same key: 4,4,4
Number of cache hits: 2
Number of cache misses: 1
*/
```

Concurrent Access

The cache is designed to be safe for concurrent access and efficient under concurrent conditions. If two concurrent processes request the same value and it is not in the cache, the value will be computed once and provided to both processes as soon as it is available. Concurrent processes will wait for the value without blocking operating system threads.

If the lookup function fails or is interrupted, the error will be propagated to all concurrent processes waiting for the value. Failures are cached to prevent repeated computation of the same failed value. If interrupted, the key will be removed from the cache, so subsequent calls will attempt to compute the value again.

Capacity

A cache is created with a specified capacity. When the cache reaches capacity, the least recently accessed values will be removed first. The cache size may slightly exceed the specified capacity between operations.

Time To Live (TTL)

A cache can also have a specified time to live (TTL). Values older than the TTL will not be returned. The age is calculated from when the value was loaded into the cache.

Operators

In addition to `get`, Cache provides several other operators:

- **refresh**: Similar to `get`, but triggers a re-computation of the value without invalidating it, allowing requests to the associated key to be served while the value is being re-computed.
- **size**: Returns the current size of the cache. Under concurrent access, the size is approximate.
- **contains**: Checks if a value associated with a specified key exists in the cache. Under concurrent access, the result is valid as of the check time but may change immediately after.
- **invalidate**: Evicts the value associated with a specified key.
- **invalidateAll**: Evicts all values in the cache.

Introduction to Effect's Control Flow Operators

Location: 400-guides/700-control-flow

Effect is a powerful TypeScript library designed to help developers easily create complex, synchronous, and asynchronous programs.

Even though JavaScript provides built-in control flow structures, Effect offers additional control flow functions that are useful in Effect applications. In this section, we will introduce different ways to control the flow of execution.

if Expression

When working with Effect values, we can use the standard JavaScript if-then-else expressions:

```
import { Effect, Option } from "effect"

const validateWeightOption = (
  weight: number
): Effect.Effect<Option.Option<number>> => {
  if (weight >= 0) {
    return Effect.succeed(Option.some(weight))
  } else {
    return Effect.succeed(Option.none())
  }
}
```

Here we are using the Option data type to represent the absence of a valid value.

We can also handle invalid inputs by using the error channel:

```
import { Effect } from "effect"

const validateWeightOrFail = (
  weight: number
): Effect.Effect<number, string> => {
  if (weight >= 0) {
    return Effect.succeed(weight)
  } else {
    return Effect.fail(`negative input: ${weight}`)
  }
}
```

Conditional Operators

when

Instead of using `if (condition) expression`, we can use the `Effect.when` function:

```
import { Effect, Option } from "effect"

const validateWeightOption = (
  weight: number
): Effect.Effect<Option.Option<number>> =>
  Effect.succeed(weight).pipe(Effect.when(() => weight >= 0))
```

Here we are using the Option data type to represent the absence of a valid value.

If the condition evaluates to `true`, the effect inside the `Effect.when` will be executed and the result will be wrapped in a `Some`, otherwise it returns `None`:

```
import { Effect, Option } from "effect"

const validateWeightOption = (
  weight: number
): Effect.Effect<Option.Option<number>> =>
  Effect.succeed(weight).pipe(Effect.when(() => weight >= 0))

// ---cut---
Effect.runPromise(validateWeightOption(100)).then(console.log)
/*
Output:
{
  _id: "Option",
  _tag: "Some",
  value: 100
}
*/

Effect.runPromise(validateWeightOption(-5)).then(console.log)
/*
Output:
{
  _id: "Option",
  _tag: "None"
}
*/
```

If the condition itself involves an effect, we can use `Effect.whenEffect`.

For example, the following function creates a random option of an integer value:

```
import { Effect, Random } from "effect"

const randomIntOption = Random.nextInt.pipe(
  Effect.whenEffect(Random.nextBoolean)
)
```

unless

The `Effect.unless` and `Effect.unlessEffect` functions are similar to the `when*` functions, but they are equivalent to the `if (!condition) expression` construct.

if

The `Effect.if` function allows you to provide an effectful predicate. If the predicate evaluates to `true`, the `onTrue` effect will be executed. Otherwise, the `onFalse` effect will be executed.

Let's use this function to create a simple virtual coin flip function:

```
import { Effect, Random, Console } from "effect"

const flipTheCoin = Effect.if(Random.nextBoolean, {
  onTrue: () => Console.log("Head"),
  onFalse: () => Console.log("Tail")
})

Effect.runPromise(flipTheCoin)
```

In this example, we generate a random boolean value using `Random.nextBoolean`. If the value is `true`, the effect `onTrue` will be executed, which logs "Head". Otherwise, if the value is `false`, the effect `onFalse` will be executed, logging "Tail".

Loop Operators

loop

The `Effect.loop` function allows you to repeatedly change the state based on an `step` function until a condition given by the `while` function is evaluated to `true`:

```
Effect.loop(initial, options: { while, step, body })
```

It collects all intermediate states in an array and returns it as the final result.

We can think of `Effect.loop` as equivalent to a `while` loop in JavaScript:

```
let state = initial
const result = []

while (options.while(state)) {
  result.push(options.body(state))
  state = options.step(state)
}

return result
```

Example

```
import { Effect } from "effect"

const result = Effect.loop(
  1, // Initial state
  {
    while: (state) => state <= 5, // Condition to continue looping
    step: (state) => state + 1, // State update function
    body: (state) => Effect.succeed(state) // Effect to be performed on each iteration
  }
)

Effect.runPromise(result).then(console.log) // Output: [1, 2, 3, 4, 5]
```

In this example, the loop starts with an initial state of `1`. The loop continues as long as the condition `n <= 5` is `true`, and in each iteration, the state `n` is incremented by `1`. The effect `Effect.succeed(n)` is performed on each iteration, collecting all intermediate states in an array.

You can also use the `discard` option if you're not interested in collecting the intermediate results. It discards all intermediate states and returns `undefined` as the final result.

Example (`discard: true`)

```
import { Effect, Console } from "effect"
```

```
const result = Effect.loop(
  1, // Initial state
  {
    while: (state) => state <= 5, // Condition to continue looping,
    step: (state) => state + 1, // State update function,
    body: (state) => Console.log(`Currently at state ${state}`), // Effect to be performed on each iteration,
    discard: true
  }
)

Effect.runPromise(result).then(console.log)
/*
Output:
Currently at state 1
Currently at state 2
Currently at state 3
Currently at state 4
Currently at state 5
undefined
*/
```

In this example, the loop performs a side effect of logging the current index on each iteration, but it discards all intermediate results. The final result is `undefined`.

iterate

The `Effect.iterate` function allows you to iterate with an effectful operation. It uses an effectful `body` operation to change the state during each iteration and continues the iteration as long as the `while` function evaluates to `true`:

```
Effect.iterate(initial, options: { while, body })
```

We can think of `Effect.iterate` as equivalent to a `while` loop in JavaScript:

```
let result = initial

while (options.while(result)) {
  result = options.body(result)
}

return result
```

Here's an example of how it works:

```
import { Effect } from "effect"

const result = Effect.iterate(
  1, // Initial result
  {
    while: (result) => result <= 5, // Condition to continue iterating
    body: (result) => Effect.succeed(result + 1) // Operation to change the result
  }
)

Effect.runPromise(result).then(console.log) // Output: 6
```

forEach

The `Effect.forEach` function allows you to iterate over an `Iterable` and perform an effectful operation for each element.

The syntax for `forEach` is as follows:

```
import { Effect } from "effect"

const combinedEffect = Effect.forEach(iterable, operation, options)
```

It applies the given effectful operation to each element of the `Iterable`. By default, it executes each effect in **sequence** (to explore options for managing concurrency and controlling how these effects are executed, you can refer to the Concurrency Options documentation).

This function returns a new effect that produces an array containing the results of each individual effect.

Let's take a look at an example:

```
import { Effect, Console } from "effect"

const result = Effect.forEach([1, 2, 3, 4, 5], (n, index) =>
  Console.log(`Currently at index ${index}`).pipe(Effect.as(n * 2))
)

Effect.runPromise(result).then(console.log)
/*
Output:
Currently at index 0
Currently at index 1
Currently at index 2
Currently at index 3
Currently at index 4
```

```
[ 2, 4, 6, 8, 10 ]
*/
```

In this example, we have an array `[1, 2, 3, 4, 5]`, and for each element we perform an effectful operation. The output shows that the operation is executed for each element in the array, displaying the current index.

The `Effect.forEach` combinator collects the results of each effectful operation in an array, which is why the final output is `[ 2, 4, 6, 8, 10 ]`.

We also have the `discard` option, which when set to `true` discards the results of each effectful operation:

```
import { Effect, Console } from "effect"

const result = Effect.forEach(
  [1, 2, 3, 4, 5],
  (n, index) =>
    Console.log(`Currently at index ${index}`).pipe(Effect.as(n * 2)),
  { discard: true }
)

Effect.runPromise(result).then(console.log)
/*
Output:
Currently at index 0
Currently at index 1
Currently at index 2
Currently at index 3
Currently at index 4
undefined
*/
```

In this case, the output is the same, but the final result is `undefined` since the results of each effectful operation are discarded.

all

The `Effect.all` function in the Effect library is a powerful tool that allows you to merge multiple effects into a single effect, offering flexibility by working with various structured formats such as tuples, iterables, structs, and records.

The syntax for `all` is as follows:

```
import { Effect } from "effect"

const combinedEffect = Effect.all(effects, options)
```

where `effects` is a collection of individual effects that you wish to merge.

By default, the `all` function will execute each effect in **sequence** (to explore options for managing concurrency and controlling how these effects are executed, you can refer to the Concurrency Options documentation).

It will return a new effect that produces a result with a shape that depends on the shape of the `effects` argument.

Let's explore examples for each supported shape: tuples, iterables, structs, and records.

Tuples

```
import { Effect, Console } from "effect"

const tuple = [
  Effect.succeed(42).pipe(Effect.tap(Console.log)),
  Effect.succeed("Hello").pipe(Effect.tap(Console.log))
] as const

const combinedEffect = Effect.all(tuple)

Effect.runPromise(combinedEffect).then(console.log)
/*
Output:
42
Hello
[ 42, 'Hello' ]
*/
```

Iterables

```
import { Effect, Console } from "effect"

const iterable: Iterable<Effect.Effect<number>> = [1, 2, 3].map((n) =>
  Effect.succeed(n).pipe(Effect.tap(Console.log))
)

const combinedEffect = Effect.all(iterable)

Effect.runPromise(combinedEffect).then(console.log)
/*
Output:
```

```
1
2
3
[ 1, 2, 3 ]
*/
```

Structs

```
import { Effect, Console } from "effect"

const struct = {
  a: Effect.succeed(42).pipe(Effect.tap(Console.log)),
  b: Effect.succeed("Hello").pipe(Effect.tap(Console.log))
}

const combinedEffect = Effect.all(struct)

Effect.runPromise(combinedEffect).then(console.log)
/*
Output:
42
Hello
{ a: 42, b: 'Hello' }
*/
```

Records

```
import { Effect, Console } from "effect"

const record: Record<string, Effect.Effect<number>> = {
  key1: Effect.succeed(1).pipe(Effect.tap(Console.log)),
  key2: Effect.succeed(2).pipe(Effect.tap(Console.log))
}

const combinedEffect = Effect.all(record)

Effect.runPromise(combinedEffect).then(console.log)
/*
Output:
1
2
{ key1: 1, key2: 2 }
*/
```

The Role of Short-Circuiting

When working with the `Effect.all` API, it's important to understand how it manages errors. This API is designed to **short-circuit the execution** upon encountering the **first error**.

What does this mean for you as a developer? Well, let's say you have a collection of effects to be executed in sequence. If any error occurs during the execution of one of these effects, the remaining computations will be skipped, and the error will be propagated to the final result.

In simpler terms, the short-circuiting behavior ensures that if something goes wrong at any step of your program it will immediately stop and return the error to let you know that something went wrong.

```
import { Effect, Console } from "effect"

const effects = [
  Effect.succeed("Task1").pipe(Effect.tap(Console.log)),
  Effect.fail("Task2: Oh no!").pipe(Effect.tap(Console.log)),
  Effect.succeed("Task3").pipe(Effect.tap(Console.log)) // this task won't be executed
]

const program = Effect.all(effects)

Effect.runPromiseExit(program).then(console.log)
/*
Output:
Task1
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: 'Task2: Oh no!' }
}
*/
```

You can override this behavior by using the `mode` option.

The mode option

When you use the `{ mode: "either" }` option with `Effect.all`, it modifies the behavior of the API to handle errors differently. Instead of short-circuiting the entire computation on the first error, it continues to execute all effects, collecting both successes and failures. The result is an array of `Either` instances, representing either a successful outcome (`Right`) or a failure (`Left`) for each individual effect.

Here's a breakdown:

```
import { Effect, Console } from "effect"

const effects = [
  Effect.succeed("Task1").pipe(Effect.tap(Console.log)),
  Effect.fail("Task2: Oh no!").pipe(Effect.tap(Console.log)),
  Effect.succeed("Task3").pipe(Effect.tap(Console.log))
]

const program = Effect.all(effects, { mode: "either" })

Effect.runPromiseExit(program).then(console.log)
/*
Output:
Task1
Task3
{
  _id: 'Exit',
  _tag: 'Success',
  value: [
    { _id: 'Either', _tag: 'Right', right: 'Task1' },
    { _id: 'Either', _tag: 'Left', left: 'Task2: Oh no!' },
    { _id: 'Either', _tag: 'Right', right: 'Task3' }
  ]
}
*/
```

On the other hand, when you use the `{ mode: "validate" }` option with `Effect.all`, it takes a similar approach to `{ mode: "either" }` but uses the `Option` type to represent the success or failure of each effect. The resulting array will contain `None` for successful effects and `Some` with the associated error message for failed effects.

Here's an illustration:

```
import { Effect, Console } from "effect"

const effects = [
  Effect.succeed("Task1").pipe(Effect.tap(Console.log)),
  Effect.fail("Task2: Oh no!").pipe(Effect.tap(Console.log)),
  Effect.succeed("Task3").pipe(Effect.tap(Console.log))
]

const program = Effect.all(effects, { mode: "validate" })

Effect.runPromiseExit(program).then((result) => console.log("%o", result))
/*
Output:
Task1
Task3
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: {
    _id: 'Cause',
    _tag: 'Fail',
    failure: [
      { _id: 'Option', _tag: 'None' },
      { _id: 'Option', _tag: 'Some', value: 'Task2: Oh no!' },
      { _id: 'Option', _tag: 'None' }
    ]
  }
}
*/
```

Introduction to Metrics in Effect

Location: 400-guides/400-observability/400-telemetry/200-metrics

Effect Metrics provides a powerful solution for monitoring and analyzing various metrics, offering support for counters, gauges, histograms, summaries, and frequencies. Learn how these metrics enhance visibility into your application's performance and behavior.

In complex and highly concurrent applications, managing various interconnected components can be quite challenging. Ensuring that everything runs smoothly and avoiding application downtime becomes crucial in such setups.

Now, let's imagine we have a sophisticated infrastructure with numerous services. These services are replicated and distributed across our servers. However, we often lack insight into what's happening across these services, including error rates, response times, and service uptime. This lack of visibility can make it challenging to identify and address issues effectively. This is where Effect Metrics comes into play; it allows us to capture and analyze various metrics, providing valuable data for later investigation.

Effect Metrics offers support for five different types of metrics:

1. **Counter**: Counters are used to track values that increase over time, such as request counts. They help us keep tabs on how many times a specific event or action has occurred.

2. **Gauge**: Gauges represent a single numerical value that can fluctuate up and down over time. They are often used to monitor metrics like memory usage, which can vary continuously.

3. **Histogram**: Histograms are useful for tracking the distribution of observed values across different buckets. They are commonly used for metrics like request latencies, allowing us to understand how response times are distributed.

4. **Summary**: Summaries provide insight into a sliding window of a time series and offer metrics for specific percentiles of the time series, often referred to as quantiles. This is particularly helpful for understanding latency-related metrics, such as request response times.

5. **Frequency**: Frequency metrics count the occurrences of distinct string values. They are useful when you want to keep track of how often different events or conditions are happening in your application.

Counter

In the world of metrics, a Counter is a metric that represents a single numerical value that can be both incremented and decremented over time. Think of it like a tally that keeps track of changes, such as the number of a particular type of request received by your application, whether it's increasing or decreasing.

Unlike some other types of metrics (like gauges), where we're interested in the value at a specific moment, with counters, we care about the cumulative value over time. This means it provides a running total of changes, which can go up and down, reflecting the dynamic nature of certain metrics.

How to Create a Counter

To create a counter, you can use the `Metric.counter` constructor in your code. You have the option to specify the type of the counter as either `number` or `bigint`. Here's how you can do it:

```
import { Metric } from "effect"

const numberCounter = Metric.counter("request_count", {
  description: "A counter for tracking requests"
})

const bigintCounter = Metric.counter("error_count", {
  description: "A counter for tracking errors",
  bigint: true
})
```

If you wish to create a counter that only increases its value, you can utilize the `incremental: true` option as follows:

```
import { Metric } from "effect"

const incrementalCounter = Metric.counter("count", {
  description: "a counter that only increases its value",
  incremental: true
})
```

With this configuration, Effect ensures that non-incremental updates have no impact on the counter, making it exclusively suitable for counting upwards.

When to Use Counters

Counters are incredibly useful when you need to keep track of cumulative values that can both increase and decrease over time. So, when should you use counters?

1. **Tracking a Value Over Time**: If you need to monitor something that consistently increases over time, like the number of incoming requests, counters are your go-to choice.

2. **Measuring Growth Rates**: Counters are also handy when you want to measure how fast something is growing. For instance, you can use them to keep tabs on request rates.

Counters find application in various scenarios, including:

- **Request Counts**: Monitoring the number of incoming requests to your server.
- **Completed Tasks**: Keeping track of how many tasks or processes have been successfully completed.
- **Error Counts**: Counting the occurrences of errors in your application.

Example

Here's a practical example of creating and using a counter in your code:

```
import { Metric, Effect, Console } from "effect"

// Create a counter named 'task_count' and increment it by 1 every time it's invoked
const taskCount = Metric.counter("task_count").pipe(
  Metric.withConstantInput(1)
)
```

```
const task1 = Effect.succeed(1).pipe(Effect.delay("100 millis"))
const task2 = Effect.succeed(2).pipe(Effect.delay("200 millis"))

const program = Effect.gen(function* () {
  const a = yield* taskCount(task1)
  const b = yield* taskCount(task2)
  return a + b
})

const showMetric = Metric.value(taskCount).pipe(Effect.andThen(Console.log))

Effect.runPromise(program.pipe(Effect.tap(() => showMetric))).then(
  console.log
)
/*
Output:
CounterState {
  count: 2,
  ...
}
3
*/
```

In this example, we create a counter called `taskCount`, which is incremented by 1 each time it's invoked. We then use it to monitor the number of times certain tasks are executed. The result provides valuable insights into the cumulative count of these tasks.

It's worth noting that applying the `taskCount` metric to an effect doesn't change its type. So, if `task1` has a type of `Effect<number>`, then `taskCount(task1)` still has the same type, `Effect<number>`.

Gauge

In the world of metrics, a Gauge is a metric that represents a single numerical value that can be set or adjusted. Think of it as a dynamic variable that can change over time. One common use case for a gauge is to monitor something like the current memory usage of your application.

Unlike counters, where we're interested in cumulative values over time, with gauges, our focus is on the current value at a specific point in time.

How to Create a Gauge

To create a gauge, you can use the `Metric.gauge` constructor in your code. You can specify the type of the gauge as either `number` or `bigint`. Here's how you can do it:

```
import { Metric } from "effect"

const numberGauge = Metric.gauge("memory_usage", {
  description: "A gauge for memory usage"
})

const bigintGauge = Metric.gauge("cpu_load", {
  description: "A gauge for CPU load",
  bigint: true
})
```

When to Use Gauges

Gauges are the best choice when you want to monitor values that can both increase and decrease, and you're not interested in tracking their rates of change. In other words, gauges help us measure things that have a specific value at a particular moment:

- **Memory Usage**: Keeping an eye on how much memory your application is using right now.
- **Queue Size**: Monitoring the current size of a queue where tasks are waiting to be processed.
- **In-Progress Request Counts**: Tracking the number of requests currently being handled by your server.
- **Temperature**: Measuring the current temperature, which can fluctuate up and down.

Example

Let's look at a practical example of creating and using a gauge in your code:

```
import { Metric, Effect, Random, Console } from "effect"

const temperature = Metric.gauge("temperature")

const getTemperature = Effect.gen(function* () {
  const n = yield* Random.nextIntBetween(-10, 10)
  console.log(`variation: ${n}`)
  return n
})

const program = Effect.gen(function* () {
  const series: Array<number> = []
  series.push(yield* temperature(getTemperature))
```

```
  series.push(yield* temperature(getTemperature))
  series.push(yield* temperature(getTemperature))
  return series
})

const showMetric = Metric.value(temperature).pipe(Effect.andThen(Console.log))

Effect.runPromise(program.pipe(Effect.tap(() => showMetric))).then(
  console.log
)
/*
Output:
variation: 6
variation: -4
variation: -9
GaugeState {
  value: -9,
  ...
}
[ 6, -4, -9 ]
*/
```

Histogram

A Histogram is a metric that helps us understand how a collection of numerical values is distributed over time. Instead of just focusing on the individual values, histograms organize these values into distinct intervals, called buckets, and record the frequency of values within each bucket.

Histograms are valuable because they not only represent the actual values but also provide insights into their distribution. They are like a summary of a dataset, breaking down the data into buckets and showing how many data points fall into each one.

How Histograms Work

In a histogram, each incoming sample is assigned to a predefined bucket. When a data point arrives, it increases the count for the corresponding bucket, and then the individual sample is discarded. This bucketed approach allows us to aggregate data across multiple instances. Histograms are especially useful for measuring percentiles, helping us estimate specific percentiles by looking at bucket counts.

Key Concepts

- **Observing Values:** Histograms observe numerical values and count how many observations fall into specific buckets. Each bucket has an upper boundary, and the count for a bucket increases by 1 if an observed value is less than or equal to the bucket's upper boundary.
- **Overall Count:** A histogram also keeps track of the total count of observed values and the sum of all observed values.
- **Inspired by Prometheus:** The concept of histograms is inspired by Prometheus, a popular monitoring and alerting toolkit.

When to Use Histograms

Histograms are widely used in software metrics for various purposes, especially in analyzing the performance of software systems. They are valuable for metrics such as response times, latencies, and throughput. By visualizing the distribution of these metrics in a histogram, developers can identify performance bottlenecks, outliers, or variations. This information helps in optimizing code, infrastructure, and system configurations to improve overall performance.

Histograms are the best choice in the following situations:

- When you want to observe many values and later calculate percentiles of those observed values.
- When you can estimate the range of values in advance, as histograms organize observations into predefined buckets.
- When you don't require exact values due to the inherent lossy nature of bucketing data in histograms.
- When you need to aggregate histograms across multiple instances.

Examples

Histogram With Linear Buckets

In this example, we create a histogram with linear buckets, ranging from 0 to 100 in increments of 10, and an "Infinity" bucket. It's suitable for effects yielding a `number`. The program then generates random values, records them in the histogram, and displays the histogram's state.

```
import { Effect, Metric, MetricBoundaries, Random } from "effect"

const latencyHistogram = Metric.histogram(
  "request_latency",
  MetricBoundaries.linear({ start: 0, width: 10, count: 11 })
)

const program = latencyHistogram(Random.nextIntBetween(1, 120)).pipe(
  Effect.repeatN(99)
)
```

```
Effect.runPromise(
  program.pipe(Effect.andThen(Metric.value(latencyHistogram)))
).then((histogramState) => console.log("%o", histogramState))
/*
Output:
HistogramState {
  buckets: [
    [ 0, 0 ],
    [ 10, 7 ],
    [ 20, 11 ],
    [ 30, 20 ],
    [ 40, 27 ],
    [ 50, 38 ],
    [ 60, 53 ],
    [ 70, 64 ],
    [ 80, 73 ],
    [ 90, 84 ],
    [ Infinity, 100 ],
    [length]: 11
  ],
  count: 100,
  min: 1,
  max: 119,
  sum: 5980,
  ...
}
*/
```

Timer Metric

This example demonstrates the use of a timer metric to track workflow durations. It generates random values, simulates waiting times, records durations in the timer metric, and displays the histogram's state.

```
import { Metric, Array, Random, Effect } from "effect"

// Metric<Histogram, Duration, Histogram>
const timer = Metric.timerWithBoundaries("timer", Array.range(1, 10))

const program = Random.nextIntBetween(1, 10).pipe(
  Effect.andThen((n) => Effect.sleep(`${n} millis`)),
  Metric.trackDuration(timer),
  Effect.repeatN(99)
)

Effect.runPromise(program.pipe(Effect.andThen(Metric.value(timer)))).then(
  (histogramState) => console.log("%o", histogramState)
)
/*
Output:
HistogramState {
  buckets: [
    [ 1, 3 ],
    [ 2, 13 ],
    [ 3, 17 ],
    [ 4, 26 ],
    [ 5, 35 ],
    [ 6, 43 ],
    [ 7, 53 ],
    [ 8, 56 ],
    [ 9, 65 ],
    [ 10, 72 ],
    [ Infinity, 100 ],
    [length]: 11
  ],
  count: 100,
  min: 0.25797,
  max: 12.25421,
  sum: 683.0266810000002,
  ...
}
*/
```

These examples showcase how histograms can be used to analyze and understand the distribution of data in various scenarios, making them a valuable tool in software metrics.

Summary

A Summary is a metric that provides valuable insights into a time series by calculating specific percentiles. These percentiles help us understand the distribution of values within the time series. Imagine you're tracking response times for requests over the past hour; you might be interested in percentiles like the 50th, 90th, 95th, and 99th to analyze performance.

How Summaries Work

Summaries, much like histograms, observe `number` values. However, instead of directly modifying bucket counters and discarding samples, summaries retain the observed samples in their internal state. To prevent uncontrolled growth of the sample set, a summary is configured with a

maximum age `maxAge` and a maximum size `maxSize`. When calculating statistics, it uses a maximum of `maxSize` samples, all of which are not older than `maxAge`.

Think of the set of samples as a sliding window over the most recent observations that meet the specified conditions.

Summaries are primarily used to calculate quantiles over the current set of samples. A quantile is defined by a `number` value `q` with `0 <= q <= 1` and results in a `number` as well.

The value of a specific quantile `q` is determined as the maximum value `v` from the current sample buffer (with size `n`) where at most `q * n` values from the sample buffer are less than or equal to `v`.

Common quantiles for observation include `0.5` (the median) and `0.95`. Quantiles are particularly useful for monitoring Service Level Agreements (SLAs).

The Effect Metrics API also allows summaries to be configured with an error margin `error`. This margin is applied to the count of values, so a quantile `q` for a set of size `s` resolves to value `v` if the count `n` of values less than or equal to `v` falls within the range `(1 - error)q * s <= n <= (1 + error)q`.

When to Use Summaries

Summaries are excellent for monitoring latencies when histograms are not the right fit due to accuracy concerns. They shine in situations where:

- The range of values is not well-estimated, making histograms less suitable.
- There's no need for aggregation or averaging across multiple instances, as summary calculations are performed on the application side.

Example

Let's create a summary to hold `100` samples, with a maximum sample age of `1 day`, and an error margin of `3%`. This summary should report the `10%`, `50%`, and `90%` quantiles. It can be applied to effects yielding integers:

```
import { Metric, Random, Effect } from "effect"

const responseTimeSummary = Metric.summary({
  name: "response_time_summary",
  maxAge: "1 day",
  maxSize: 100,
  error: 0.03,
  quantiles: [0.1, 0.5, 0.9]
})

const program = responseTimeSummary(Random.nextIntBetween(1, 120)).pipe(
  Effect.repeatN(99)
)

Effect.runPromise(
  program.pipe(Effect.andThen(Metric.value(responseTimeSummary)))
).then((summaryState) => console.log("%o", summaryState))
/*
Output:
SummaryState {
  error: 0.03,
  quantiles: [
    [ 0.1, { _id: 'Option', _tag: 'Some', value: 17 } ],
    [ 0.5, { _id: 'Option', _tag: 'Some', value: 62 } ],
    [ 0.9, { _id: 'Option', _tag: 'Some', value: 109 } ]
  ],
  count: 100,
  min: 4,
  max: 119,
  sum: 6058,
  ...
}
*/
```

Frequency

Frequencies are metrics that help us count the occurrences of specific values. Think of them as a set of counters, each associated with a unique value. When new values are observed, frequencies automatically create new counters for them.

When to Use Frequencies

Frequencies are invaluable for counting the occurrences of distinct string values. Consider using frequencies in scenarios like:

- Tracking the number of invocations for each service in an application that uses logical names for its services.
- Monitoring the frequency of different types of failures.

Example

Let's create a Frequency to observe the occurrences of unique strings. This example can be applied to effects that yield a `string`:

```
import { Metric, Random, Effect } from "effect"

const errorFrequency = Metric.frequency("error_frequency")

const program = errorFrequency(
  Random.nextIntBetween(1, 10).pipe(Effect.andThen((n) => `Error-${n}`))
).pipe(Effect.repeatN(99))

Effect.runPromise(
  program.pipe(Effect.andThen(Metric.value(errorFrequency)))
).then((frequencyState) => console.log("%o", frequencyState))
/*
Output:
FrequencyState {
  occurrences: Map(9) {
    'Error-7' => 12,
    'Error-2' => 12,
    'Error-4' => 14,
    'Error-1' => 14,
    'Error-9' => 8,
    'Error-6' => 11,
    'Error-5' => 9,
    'Error-3' => 14,
    'Error-8' => 6
  },
  ...
}
*/
```

Telemetry

Location: 400-guides/400-observability/400-telemetry/index

Telemetry

Introduction to Tracing in Effect

Location: 400-guides/400-observability/400-telemetry/300-tracing

Explore the necessity of tracing in distributed systems beyond logs and metrics. Discover spans and traces, crucial for understanding request lifecycles. Learn to create, print, and annotate spans, and visualize traces for effective debugging and optimization.

Although logs and metrics are useful to understand the behavior of individual services, they are not enough to provide a complete overview of the lifetime of a request in a distributed system.

In a distributed system, a request can span multiple services and each service can make multiple requests to other services to fulfill the request. In such a scenario, we need to have a way to track the lifetime of a request across multiple services to diagnose what services are the bottlenecks and where the request is spending most of its time.

Spans

A span represents a unit of work or operation. It tracks specific operations that a request makes, painting a picture of what happened during the time in which that operation was executed.

A typical Span contains the following information:

- **Name:** Describes the operation being tracked.
- **Time-Related Data:** Timestamps to measure when the operation started and how long it took.
- **Structured Log Messages:** Records essential information during the operation.
- **Metadata (Attributes):** Additional data that provides context about the operation.

Traces

A trace records the paths taken by requests (made by an application or end-user) as they propagate through multi-service architectures, like microservice and serverless applications.

Without tracing, it is challenging to pinpoint the cause of performance problems in a distributed system.

A trace is made of one or more spans. The first span represents the root span. Each root span represents a request from start to finish. The spans underneath the parent provide a more in-depth context of what occurs during a request (or what steps make up a request).

Many Observability back-ends visualize traces as waterfall diagrams that may look something like this:

Waterfall diagrams show the parent-child relationship between a root span and its child spans. When a span encapsulates another span, this also represents a nested relationship.

Creating Spans

You can instrument an effect with a Span using the `Effect.withSpan` API. Here's how you can do it:

```
import { Effect } from "effect"

const program = Effect.void.pipe(Effect.delay("100 millis"))

const instrumented = program.pipe(Effect.withSpan("myspan"))
```

It's important to note that instrumenting an effect doesn't change its type. You start with an `Effect<void>`, and you still get an `Effect<void>`.

Printing Spans

Now, let's print our Span to the console. To achieve this, we need specific tools, including

- `@effect/opentelemetry`
- `@opentelemetry/sdk-metrics`
- `@opentelemetry/sdk-trace-base`
- `@opentelemetry/sdk-trace-node`
- `@opentelemetry/sdk-trace-web`

With these in place, we can visualize and understand the Spans in our application.

Here's a code snippet demonstrating how to set up the necessary environment and print the Span to the console:

```
import { Effect } from "effect"
import { NodeSdk } from "@effect/opentelemetry"
import {
  ConsoleSpanExporter,
  BatchSpanProcessor
} from "@opentelemetry/sdk-trace-base"

const program = Effect.void.pipe(Effect.delay("100 millis"))

const instrumented = program.pipe(Effect.withSpan("myspan"))

const NodeSdkLive = NodeSdk.layer(() => ({
  resource: { serviceName: "example" },
  spanProcessor: new BatchSpanProcessor(new ConsoleSpanExporter())
}))

Effect.runPromise(instrumented.pipe(Effect.provide(NodeSdkLive)))
/*
Example Output:
{
  traceId: 'd0f730abfc366205806469596092b239',
  parentId: undefined,
  traceState: undefined,
  name: 'myspan',
  id: 'ab4e42592e7f1f7c',
  kind: 0,
  timestamp: 1697040012664380.5,
  duration: 2895.769,
  attributes: {},
  status: { code: 1 },
  events: [],
  links: []
}
*/
```

Here's a breakdown of the output:

Field	Description
`traceId`	A unique identifier for the entire trace, helping trace requests or operations as they move through an application.
`parentId`	Identifies the parent span of the current span, marked as `undefined` in the output when there is no parent span, making it a root span.
`name`	Describes the name of the span, indicating the operation being tracked (e.g., "myspan").
`id`	A unique identifier for the current span, distinguishing it from other spans within a trace.
`timestamp`	A timestamp representing when the span started, measured in microseconds since the Unix epoch.
`duration`	Specifies the duration of the span, representing the time taken to complete the operation (e.g., `2895.769` microseconds).
`attributes`	Spans may contain attributes, which are key-value pairs providing additional context or information about the operation. In this output, it's an empty object, indicating no specific attributes in this span.
`status`	The status field provides information about the span's status. In this case, it has a code of 1, which typically indicates an OK status (whereas a code of 2 signifies an ERROR status)
`events`	Spans can include events, which are records of specific moments during the span's lifecycle. In this output, it's an empty array, suggesting no specific events recorded.

Field	Description
`links`	Links can be used to associate this span with other spans in different traces. In the output, it's an empty array, indicating no specific links for this span.

Let's examine the output of an effect that encountered an error:

```
import { Effect } from "effect"
import { NodeSdk } from "@effect/opentelemetry"
import {
  ConsoleSpanExporter,
  BatchSpanProcessor
} from "@opentelemetry/sdk-trace-base"

const program = Effect.fail("Oh no!").pipe(
  Effect.delay("100 millis"),
  Effect.withSpan("myspan")
)

const NodeSdkLive = NodeSdk.layer(() => ({
  resource: { serviceName: "example" },
  spanProcessor: new BatchSpanProcessor(new ConsoleSpanExporter())
}))

Effect.runPromiseExit(program.pipe(Effect.provide(NodeSdkLive))).then(
  console.log
)
/*
Example Output:
{
  traceId: '760510a3f9a0881a09de990c87ec1cef',
  parentId: undefined,
  traceState: undefined,
  name: 'myspan',
  id: 'a528e38e82e848a5',
  kind: 0,
  timestamp: 1697091363002970.5,
  duration: 110371.664,
  attributes: {},
  status: { code: 2, message: 'Error: Oh no!' },
  events: [],
  links: []
}
{
  _id: 'Exit',
  _tag: 'Failure',
  cause: { _id: 'Cause', _tag: 'Fail', failure: 'Oh no!' }
}
*/
```

Adding Annotations

You can provide extra information to a span by utilizing the `Effect.annotateCurrentSpan` function. This tool allows you to associate key-value pairs, offering more context about the execution of the span.

```
import { Effect } from "effect"
import { NodeSdk } from "@effect/opentelemetry"
import {
  ConsoleSpanExporter,
  BatchSpanProcessor
} from "@opentelemetry/sdk-trace-base"

const program = Effect.void.pipe(
  Effect.delay("100 millis"),
  Effect.tap(() => Effect.annotateCurrentSpan("key", "value")),
  Effect.withSpan("myspan")
)

const NodeSdkLive = NodeSdk.layer(() => ({
  resource: { serviceName: "example" },
  spanProcessor: new BatchSpanProcessor(new ConsoleSpanExporter())
}))

Effect.runPromise(program.pipe(Effect.provide(NodeSdkLive)))
/*
Example Output:
{
  traceId: '869c9d74d9db14a4ba4393ca8e0f61db',
  parentId: undefined,
  traceState: undefined,
  name: 'myspan',
  id: '31eb49570d197f8d',
  kind: 0,
  timestamp: 1697045981663321.5,
  duration: 109563.353,
  attributes: { key: 'value' },
  status: { code: 1 },
  events: [],
```

```
  links: []
}
*/
```

Logs as events

Logs are transformed into what are known as "Span Events." These events provide structured information and a timeline of occurrences within your application.

```
import { Effect } from "effect"
import { NodeSdk } from "@effect/opentelemetry"
import {
  ConsoleSpanExporter,
  BatchSpanProcessor
} from "@opentelemetry/sdk-trace-base"

const program = Effect.log("Hello").pipe(
  Effect.delay("100 millis"),
  Effect.withSpan("myspan")
)

const NodeSdkLive = NodeSdk.layer(() => ({
  resource: { serviceName: "example" },
  spanProcessor: new BatchSpanProcessor(new ConsoleSpanExporter())
}))

Effect.runPromise(program.pipe(Effect.provide(NodeSdkLive)))
/*
Example Output:
{
  traceId: 'ad708d58c15f9e5c7b5cca2eeb6838a2',
  parentId: undefined,
  traceState: undefined,
  name: 'myspan',
  id: '4353fd47423e786a',
  kind: 0,
  timestamp: 1697043230170724.2,
  duration: 112052.514,
  attributes: {},
  status: { code: 1 },
  events: [
    {
      name: 'Hello',
      attributes: { 'effect.fiberId': '#0', 'effect.logLevel': 'INFO' },
      time: [ 1697043230, 280923805 ],
      droppedAttributesCount: 0
    }
  ],
  links: []
}
*/
```

Spans can contain events, which are records of specific moments during the span's lifecycle. In this output, there is one event named `'Hello'`. It includes associated attributes, such as `'effect.fiberId'` and `'effect.logLevel'`, providing information about the logged event. The `time` field represents the timestamp when the event occurred.

Nesting Spans

Spans can be nested, creating a hierarchy of operations. This concept is illustrated in the following code:

```
import { Effect } from "effect"
import { NodeSdk } from "@effect/opentelemetry"
import {
  ConsoleSpanExporter,
  BatchSpanProcessor
} from "@opentelemetry/sdk-trace-base"

const child = Effect.void.pipe(
  Effect.delay("100 millis"),
  Effect.withSpan("child")
)

const parent = Effect.gen(function* () {
  yield* Effect.sleep("20 millis")
  yield* child
  yield* Effect.sleep("10 millis")
}).pipe(Effect.withSpan("parent"))

const NodeSdkLive = NodeSdk.layer(() => ({
  resource: { serviceName: "example" },
  spanProcessor: new BatchSpanProcessor(new ConsoleSpanExporter())
}))

Effect.runPromise(parent.pipe(Effect.provide(NodeSdkLive)))
/*
Example Output:
```

```
{
  traceId: '92fe81f1454d9c099198568cf867dc59',
  parentId: 'b953d6c7d37ad93d',
  traceState: undefined,
  name: 'child',
  id: '2fd19c8c23ebc7e8',
  kind: 0,
  timestamp: 1697043815321888.2,
  duration: 106536.264,
  attributes: {},
  status: { code: 1 },
  events: [],
  links: []
}
{
  traceId: '92fe81f1454d9c099198568cf867dc59',
  parentId: undefined,
  traceState: undefined,
  name: 'parent',
  id: 'b953d6c7d37ad93d',
  kind: 0,
  timestamp: 1697043815292133.2,
  duration: 149724.295,
  attributes: {},
  status: { code: 1 },
  events: [],
  links: []
}
*/
```

As you can observe, the `b953d6c7d37ad93d` value plays a crucial role in maintaining the parent-child relationship between these spans. It provides a clear indication of how spans can be nested, creating a trace that helps developers understand the flow and hierarchy of operations in their applications.

Tutorial: Visualizing Traces with Docker, Prometheus, Grafana, and Tempo

In this tutorial, we'll guide you through simulating and visualizing traces using a sample instrumented Node.js application. We will use Docker, Prometheus, Grafana, and Tempo to create, collect, and visualize traces.

Tools Explained

Let's understand the tools we'll be using in simple terms:

- **Docker**: Docker allows us to run applications in containers. Think of a container as a lightweight and isolated environment where your application can run consistently, regardless of the host system. It's a bit like a virtual machine but more efficient.
- **Prometheus**: Prometheus is a monitoring and alerting toolkit. It collects metrics and data about your applications and stores them for further analysis. This helps in identifying performance issues and understanding the behavior of your applications.
- **Grafana**: Grafana is a visualization and analytics platform. It helps in creating beautiful and interactive dashboards to visualize your application's data. You can use it to graphically represent metrics collected by Prometheus.
- **Tempo**: Tempo is a distributed tracing system that allows you to trace the journey of a request as it flows through your application. It provides insights into how requests are processed and helps in debugging and optimizing your applications.

Getting Docker

To get Docker, follow these steps:

1. Visit the Docker website at https://www.docker.com/.
2. Download Docker Desktop for your operating system (Windows or macOS) and install it.
3. After installation, open Docker Desktop, and it will run in the background.

Simulating Traces

Now, let's simulate traces using a sample Node.js application. We'll provide you with the code and guide you on setting up the necessary components.

1. **Download Docker Files**. Download the required Docker files: docker.zip.
2. **Set Up docker**. Unzip the downloaded file, navigate to the `/docker/local` directory in your terminal or command prompt and run the following command to start the necessary services:

   ```
   docker-compose up
   ```

3. **Simulate Traces**. Run the following example code in your Node.js environment. This code simulates a set of tasks and generates traces.

   Before proceeding, you'll need to install additional libraries in addition to the latest version of `effect`. Here are the required libraries:

    - `@effect/opentelemetry`
    - `@opentelemetry/exporter-trace-otlp-http`
    - `@opentelemetry/sdk-trace-node`
    - `@opentelemetry/sdk-trace-web`

```
import { Effect } from "effect"
import { NodeSdk } from "@effect/opentelemetry"
import { BatchSpanProcessor } from "@opentelemetry/sdk-trace-base"
import { OTLPTraceExporter } from "@opentelemetry/exporter-trace-otlp-http"

// Function to simulate a task with possible subtasks
const task = (
  name: string,
  delay: number,
  children: ReadonlyArray<Effect.Effect<void>> = []
) =>
  Effect.gen(function* () {
    yield* Effect.log(name)
    yield* Effect.sleep(`${delay} millis`)
    for (const child of children) {
      yield* child
    }
    yield* Effect.sleep(`${delay} millis`)
  }).pipe(Effect.withSpan(name))

const poll = task("/poll", 1)

// Create a program with tasks and subtasks
const program = task("client", 2, [
  task("/api", 3, [
    task("/authN", 4, [task("/authZ", 5)]),
    task("/payment Gateway", 6, [task("DB", 7), task("Ext. Merchant", 8)]),
    task("/dispatch", 9, [
      task("/dispatch/search", 10),
      Effect.all([poll, poll, poll], { concurrency: "inherit" }),
      task("/pollDriver/{id}", 11)
    ])
  ])
])

const NodeSdkLive = NodeSdk.layer(() => ({
  resource: { serviceName: "example" },
  spanProcessor: new BatchSpanProcessor(new OTLPTraceExporter())
}))

Effect.runPromise(
  program.pipe(
    Effect.provide(NodeSdkLive),
    Effect.catchAllCause(Effect.logError)
  )
)
/*
Output:
timestamp=... level=INFO fiber=#0 message=client
timestamp=... level=INFO fiber=#0 message=/api
timestamp=... level=INFO fiber=#0 message=/authN
timestamp=... level=INFO fiber=#0 message=/authZ
timestamp=... level=INFO fiber=#0 message="/payment Gateway"
timestamp=... level=INFO fiber=#0 message=DB
timestamp=... level=INFO fiber=#0 message="Ext. Merchant"
timestamp=... level=INFO fiber=#0 message=/dispatch
timestamp=... level=INFO fiber=#0 message=/dispatch/search
timestamp=... level=INFO fiber=#3 message=/poll
timestamp=... level=INFO fiber=#4 message=/poll
timestamp=... level=INFO fiber=#5 message=/poll
timestamp=... level=INFO fiber=#0 message=/pollDriver/{id}
*/
```

4. **Visualize Traces**. Now, open your web browser and go to http://localhost:3000/explore. You will see a generated `Trace ID` on the web page. Click on it to see the details of the trace.

    
    Traces in Grafana Tempo

That's it! You've simulated and visualized traces using Docker, Prometheus, Grafana, and Tempo. You can use these tools to monitor, analyze, and gain insights into your applications' performance and behavior.

Introduction to Telemetry in Effect

Location: 400-guides/400-observability/400-telemetry/100-intro

Explore the seamless integration of Telemetry in Effect, providing insights into metrics and traces. Learn how Telemetry empowers developers to monitor, analyze, and optimize the performance and behavior of TypeScript applications, enhancing observability and debugging capabilities.

Telemetry enables developers to collect, process, and export telemetry data such as metrics and traces, providing valuable insights into the performance and behavior of their applications. In this guide, we will explore how Telemetry can be seamlessly integrated into Effect.

By incorporating Telemetry, you can gain deep visibility into the inner workings of your TypeScript applications, effortlessly tracing the flow of requests, pinpointing bottlenecks, and identifying performance optimizations. This integration empowers you to monitor, analyze, and improve your application's reliability and efficiency, ensuring a better user experience.

In the following sections, we will delve into the fundamentals of Telemetry, how it works within the Effect framework, and the benefits it brings to your TypeScript projects. Whether you are building a small-scale application or a complex distributed system, this guide will help you harness the power of Telemetry to enhance your observability and debugging capabilities in Effect.

Supervisor

Location: 400-guides/400-observability/200-supervisor

Learn about Effect's `Supervisor` for managing fibers, creating supervisors with `Supervisor.track`, and supervising effects with `Effect.supervised`. Explore an example that demonstrates periodic monitoring of fibers throughout an application's lifecycle using supervisors.

A `Supervisor<A>` is a tool that manages the creation and termination of fibers, producing some visible value of type `A` based on its supervision.

Creating

track

To create a supervisor, you can use the `Supervisor.track` function. It generates a new supervisor that keeps track of its child fibers in a set.

Supervising

supervised

Whenever you need to supervise an effect, you can employ the `Effect.supervised` function. This function takes a supervisor and returns an effect that behaves the same as the original effect. However, all child fibers forked within this effect are reported to the specified supervisor.

By doing this, you associate the behavior of child fibers with the provided supervisor, giving you access to all the information about these child fibers through the supervisor.

Example

In the following example, we will periodically monitor the number of fibers throughout our application's lifecycle:

```
import { Effect, Supervisor, Schedule, Fiber, FiberStatus } from "effect"

const program = Effect.gen(function* () {
  const supervisor = yield* Supervisor.track
  const fibFiber = yield* fib(20).pipe(
    Effect.supervised(supervisor),
    Effect.fork
  )
  const policy = Schedule.spaced("500 millis").pipe(
    Schedule.whileInputEffect((_) =>
      Fiber.status(fibFiber).pipe(
        Effect.andThen((status) => status !== FiberStatus.done)
      )
    )
  )
  const monitorFiber = yield* monitorFibers(supervisor).pipe(
    Effect.repeat(policy),
    Effect.fork
  )
  yield* Fiber.join(monitorFiber)
  const result = yield* Fiber.join(fibFiber)
  console.log(`fibonacci result: ${result}`)
})

const monitorFibers = (
  supervisor: Supervisor.Supervisor<Array<Fiber.RuntimeFiber<any, any>>>
): Effect.Effect<void> =>
  Effect.gen(function* () {
    const fibers = yield* supervisor.value
    console.log(`number of fibers: ${fibers.length}`)
  })

const fib = (n: number): Effect.Effect<number> =>
  Effect.gen(function* () {
    if (n <= 1) {
      return 1
    }
    yield* Effect.sleep("500 millis")
    const fiber1 = yield* Effect.fork(fib(n - 2))
    const fiber2 = yield* Effect.fork(fib(n - 1))
    const v1 = yield* Fiber.join(fiber1)
    const v2 = yield* Fiber.join(fiber2)
```

```
    return v1 + v2
  })

Effect.runPromise(program)
/*
Output:
number of fibers: 0
number of fibers: 2
number of fibers: 6
number of fibers: 14
number of fibers: 30
number of fibers: 62
number of fibers: 126
number of fibers: 254
number of fibers: 510
number of fibers: 1022
number of fibers: 2034
number of fibers: 3795
number of fibers: 5810
number of fibers: 6474
number of fibers: 4942
number of fibers: 2515
number of fibers: 832
number of fibers: 170
number of fibers: 18
number of fibers: 0
fibonacci result: 10946
*/
```

Observability

Location: 400-guides/400-observability/index

Observability

Logging

Location: 400-guides/400-observability/100-logging

Explore the power of logging in Effect for enhanced debugging and monitoring. Learn about dynamic log level control, custom logging output, fine-grained logging, environment-based logging, and additional features. Dive into specific logging utilities such as log, logDebug, logInfo, logWarning, logError, logFatal, and spans. Discover how to disable default logging and load log levels from configuration. Finally, explore the creation of custom loggers to tailor logging to your needs.

Logging is a crucial aspect of software development, especially when it comes to debugging and monitoring the behavior of your applications. In this section, we'll delve into Effect's logging utilities and explore their advantages over traditional methods like `console.log`.

Advantages Over Traditional Logging

Effect's logging utilities offer several advantages over traditional logging methods like `console.log`:

1. **Dynamic Log Level Control**: With Effect's logging, you have the ability to change the log level dynamically. This means you can control which log messages get displayed based on their severity. For example, you can configure your application to log only warnings or errors, which can be extremely helpful in production environments to reduce noise.

2. **Custom Logging Output**: Effect's logging utilities allow you to change how logs are handled. You can direct log messages to various destinations, such as a service or a file, using a custom logger. This flexibility ensures that logs are stored and processed in a way that best suits your application's requirements.

3. **Fine-Grained Logging**: Effect enables fine-grained control over logging on a per-part basis of your program. You can set different log levels for different parts of your application, tailoring the level of detail to each specific component. This can be invaluable for debugging and troubleshooting, as you can focus on the information that matters most.

4. **Environment-Based Logging**: Effect's logging utilities can be combined with deployment environments to achieve granular logging strategies. For instance, during development, you might choose to log everything at a trace level and above for detailed debugging. In contrast, your production version could be configured to log only errors or critical issues, minimizing the impact on performance and noise in production logs.

5. **Additional Features**: Effect's logging utilities come with additional features such as the ability to measure time spans, alter log levels on a per-effect basis, and integrate spans for performance monitoring.

Now, let's dive into the specific logging utilities provided by Effect.

log

The `Effect.log` function outputs a log message at the default `INFO` level.

```
import { Effect } from "effect"

const program = Effect.log("Application started")

Effect.runSync(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message="Application started"
*/
```

Details included in a log message:

- `timestamp`: The timestamp when the log message was generated.
- `level`: The log level at which the message is logged.
- `fiber`: The identifier of the fiber executing the program.
- `message`: The log content, which can include multiple items.
- `span`: (Optional) The duration of the span in milliseconds.

You can log multiple messages simultaneously:

```
import { Effect } from "effect"

const program = Effect.log("message1", "message2", "message3")

Effect.runSync(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message=message1 message=message2 message=message3
*/
```

For added context, you can also include one or more Cause instances in your logs, which provide detailed error information under an additional `cause` annotation:

```
import { Effect, Cause } from "effect"

const program = Effect.log(
  "message1",
  "message2",
  Cause.die("Oh no!"),
  Cause.die("Oh uh!")
)

Effect.runSync(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message=message1 message=message2 cause="Error: Oh no!
Error: Oh uh!"
*/
```

Log Levels

logDebug

By default, `DEBUG` messages are **not** printed.

However, you can configure the default logger to enable them using `Logger.withMinimumLogLevel` and setting the minimum log level to `LogLevel.Debug`.

Here's an example that demonstrates how to enable `DEBUG` messages for a specific task (`task1`):

In the above example, we enable `DEBUG` messages specifically for `task1` by using the `Logger.withMinimumLogLevel` function.

By using `Logger.withMinimumLogLevel(effect, level)`, you have the flexibility to selectively enable different log levels for specific effects in your program. This allows you to control the level of detail in your logs and focus on the information that is most relevant to your debugging and troubleshooting needs.

logInfo

By default, `INFO` messages are printed.

In the above example, the `Effect.log` function is used to log an `INFO` message with the content `"start"` and `"done"`. These messages will be printed during the execution of the program.

logWarning

By default, `WARN` messages are printed.

logError

By default, `ERROR` messages are printed.

logFatal

By default, `FATAL` messages are printed.

Custom Annotations

Enhance your log outputs by incorporating custom annotations with the `Effect.annotateLogs` function. This function allows you to append additional metadata to each log entry of an effect, enhancing traceability and context.

Here's how to apply a single annotation as a key/value pair:

```
import { Effect } from "effect"

const program = Effect.gen(function* () {
  yield* Effect.log("message1")
  yield* Effect.log("message2")
}).pipe(Effect.annotateLogs("key", "value")) // Annotation as key/value pair

Effect.runSync(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message=message1 key=value
timestamp=... level=INFO fiber=#0 message=message2 key=value
*/
```

To apply multiple annotations at once, you can pass an object containing several key/value pairs:

```
import { Effect } from "effect"

const program = Effect.gen(function* () {
  yield* Effect.log("message1")
  yield* Effect.log("message2")
}).pipe(Effect.annotateLogs({ key1: "value1", key2: "value2" }))

Effect.runSync(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message=message1 key2=value2 key1=value1
timestamp=... level=INFO fiber=#0 message=message2 key2=value2 key1=value1
*/
```

Annotations can also be applied with a scoped lifetime using `Effect.annotateLogsScoped`. This method confines the application of annotations to logs within a specific Scope of your effect computation:

```
import { Effect } from "effect"

const program = Effect.gen(function* () {
  yield* Effect.log("no annotations")
  yield* Effect.annotateLogsScoped({ key: "value" })
  yield* Effect.log("message1") // Annotation is applied to this log
  yield* Effect.log("message2") // Annotation is applied to this log
}).pipe(Effect.scoped, Effect.andThen(Effect.log("no annotations again")))

Effect.runSync(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message="no annotations"
timestamp=... level=INFO fiber=#0 message=message1 key=value
timestamp=... level=INFO fiber=#0 message=message2 key=value
timestamp=... level=INFO fiber=#0 message="no annotations again"
*/
```

Log Spans

Effect also provides support for log spans, allowing you to measure the duration of specific operations or tasks within your program.

In the above example, a log span is created using the `Effect.withLogSpan(label)` function. It measures the duration of the code block within the span. The resulting duration is then automatically recorded as an annotation within the log message.

Disabling Default Logging

If you ever find yourself needing to turn off default logging, perhaps during test execution, there are various ways to achieve this within the Effect framework. In this section, we'll explore different methods to disable default logging.

Using withMinimumLogLevel

Effect provides a convenient function called `withMinimumLogLevel` that allows you to set the minimum log level, effectively disabling logging:

```
import { Effect, Logger, LogLevel } from "effect"

const program = Effect.gen(function* () {
  yield* Effect.log("Executing task...")
```

```
  yield* Effect.sleep("100 millis")
  console.log("task done")
})

// Logging enabled (default)
Effect.runPromise(program)
/*
Output:
timestamp=... level=INFO fiber=#0 message="Executing task..."
task done
*/

// Logging disabled using withMinimumLogLevel
Effect.runPromise(program.pipe(Logger.withMinimumLogLevel(LogLevel.None)))
/*
Output:
task done
*/
```

By setting the log level to `LogLevel.None`, you effectively disable logging, and only the final result will be displayed.

Using a Layer

Another approach to disable logging is by creating a layer that sets the minimum log level to `LogLevel.None`, effectively turning off all logging:

```
import { Effect, Logger, LogLevel } from "effect"

const program = Effect.gen(function* () {
  yield* Effect.log("Executing task...")
  yield* Effect.sleep("100 millis")
  console.log("task done")
})

const layer = Logger.minimumLogLevel(LogLevel.None)

// Logging disabled using a layer
Effect.runPromise(program.pipe(Effect.provide(layer)))
/*
Output:
task done
*/
```

Using a Custom Runtime

You can also disable logging by creating a custom runtime that includes the configuration to turn off logging:

```
import { Effect, Logger, LogLevel, Runtime, Layer } from "effect"

const program = Effect.gen(function* () {
  yield* Effect.log("Executing task...")
  yield* Effect.sleep("100 millis")
  console.log("task done")
})

const customRuntime = Effect.runSync(
  Effect.scoped(Layer.toRuntime(Logger.minimumLogLevel(LogLevel.None)))
)

// Logging disabled using a custom runtime
const customRunPromise = Runtime.runPromise(customRuntime)

customRunPromise(program)
/*
Output:
task done
*/
```

In this approach, you create a custom runtime that incorporates the configuration to disable logging, and then you execute your program using this custom runtime.

Loading the Log Level from Configuration

To retrieve the log level from a configuration and incorporate it into your program, utilize the layer produced by `Logger.minimumLogLevel`:

```
import {
  Effect,
  Config,
  Logger,
  Layer,
  ConfigProvider,
  LogLevel
} from "effect"

// Simulate a program with logs
const program = Effect.gen(function* () {
  yield* Effect.logError("ERROR!")
  yield* Effect.logWarning("WARNING!")
```

```
  yield* Effect.logInfo("INFO!")
  yield* Effect.logDebug("DEBUG!")
})

// Load the log level from the configuration as a layer
const LogLevelLive = Config.logLevel("LOG_LEVEL").pipe(
  Effect.andThen((level) => Logger.minimumLogLevel(level)),
  Layer.unwrapEffect
)

// Configure the program with the loaded log level
const configured = Effect.provide(program, LogLevelLive)

// Test the configured program using ConfigProvider.fromMap
const test = Effect.provide(
  configured,
  Layer.setConfigProvider(
    ConfigProvider.fromMap(new Map([["LOG_LEVEL", LogLevel.Warning.label]]))
  )
)

Effect.runPromise(test)
/*
Output:
... level=ERROR fiber=#0 message=ERROR!
... level=WARN fiber=#0 message=WARNING!
*/
```

To evaluate the configured program, you can utilize `ConfigProvider.fromMap` for testing (refer to Testing Services for more details).

Custom loggers

In this section, we will learn how to define a custom logger and set it as the default logger in our program.

First, let's define our custom logger:

```
import { Logger } from "effect"

export const logger = Logger.make(({ logLevel, message }) => {
  globalThis.console.log(`[${logLevel.label}] ${message}`)
})

// @include: CustomLogger
```

Assuming we have defined the following program:

To replace the default logger, we simply need to create a specific layer using `Logger.replace` and provide it to our program using `Effect.provide` before executing it:

```
// @filename: CustomLogger.ts
// @include: CustomLogger

// @filename: program.ts
// @include: program

// @filename: index.ts
// ---cut---
import { Effect, Logger, LogLevel } from "effect"
import * as CustomLogger from "./CustomLogger"
import { program } from "./program"

const layer = Logger.replace(Logger.defaultLogger, CustomLogger.logger)

Effect.runPromise(
  Effect.provide(Logger.withMinimumLogLevel(program, LogLevel.Debug), layer)
)
```

This is what we see printed on the console after executing the program:

```
[INFO] start
[DEBUG] task1 done
[DEBUG] task2 done
[INFO] done
```

Testing

Location: 400-guides/680-testing/index

Testing

TestClock

Location: 400-guides/680-testing/100-testclock

Utilize Effect's `TestClock` to control time during testing. Learn how to simulate the passage of time and efficiently test time-related functionality. Examples include testing `Effect.timeout`, recurring effects, `Clock`, and `Deferred`.

In most cases, we want our unit tests to run as quickly as possible. Waiting for real time to pass can slow down our tests significantly. Effect provides a handy tool called `TestClock` that allows us to **control time during testing**. This means we can efficiently and predictably test code that involves time without having to wait for the actual time to pass.

The `TestClock` in Effect allows us to control time for testing purposes. It lets us schedule effects to run at specific times, making it ideal for testing time-related functionality.

Instead of waiting for real time to pass, we use the `TestClock` to set the clock time to a specific point. Any effects scheduled to run at or before that time will be executed in order.

How TestClock Works

Imagine `TestClock` as a wall clock, but with a twist—it doesn't tick on its own. Instead, it only changes when we manually adjust it using the `TestClock.adjust` and `TestClock.setTime` functions. The clock time never progresses on its own.

When we adjust the clock time, any effects scheduled to run at or before the new time will be executed. This allows us to simulate the passage of time in our tests without waiting for real time.

Let's look at an example of how to test `Effect.timeout` using the `TestClock`:

In this example, we create a test scenario involving a fiber that sleeps for 5 minutes and then times out after 1 minute. Instead of waiting for the full 5 minutes to elapse in real time, we utilize the `TestClock` to instantly advance time by 1 minute.

A critical point to note is the forking of the fiber where `Effect.sleep` is invoked. Calls to `Effect.sleep` and related methods wait until the clock time matches or surpasses the scheduled time for their execution. By forking the fiber, we ensure that we have control over the clock time adjustment.

<Idea> The recommended pattern when using the `TestClock` involves forking the effect being tested, adjusting the clock time as needed, and then verifying that the expected effects have occurred. </Idea>

More Examples

Testing Recurring Effects

Let's explore an example demonstrating how to test an effect that runs at fixed intervals using the `TestClock`:

In this example, we aim to test an effect that runs at regular intervals. We use an unbounded queue to manage the effects. We verify that:

1. No effect is performed before the specified recurrence period.
2. An effect is performed after the recurrence period.
3. The effect is executed exactly once.

It's crucial to note that after each recurrence, the next occurrence is scheduled to happen at the appropriate time in the future. When we adjust the clock by 60 minutes, precisely one value is placed in the queue, and when we adjust the clock by another 60 minutes, another value is added to the queue.

Testing Clock

Let's explore an example that demonstrates how to test the behavior of the `Clock` using the `TestClock`:

Testing Deferred

`TestClock` also affects asynchronous code scheduled to run after a certain time:

In this code, we create a scenario where a value is set in a `Deferred` after 10 seconds asynchronously. We use `Effect.fork` to run this asynchronously. By adjusting the `TestClock` by 10 seconds, we can simulate the passage of time and test the code without waiting for the actual 10 seconds to elapse.