Wifi: SALA CONVEGNI
Password: SalaConvegni

Git repository:

`https://github.com/Effect-TS/effect-days-2025-workshop.git`

Effect

Effect Days 2025

Welcome to the Workshop!

Effect

Workshop Schedule

Speaker		Time Slot	Duration
Max	Session 1	9:00 AM – 10:30 AM	1.5 hours
	Break	10:30 AM – 10:45 AM	15 minutes
	Session 2	10:45 AM – 12:15 PM	1.5 hours
	Lunch	12:15 PM – 1:15 PM	1 hour
Tim	Session 3	1:15 PM – 2:45 PM	1.5 hours
	Break	2:45 PM – 3:00 PM	15 minutes
	Session 4	3:00 PM – 4:30 PM	1.5 hours
	Q & A	4:30 PM – 5:00 PM	30 minutes

Effect

Section One

Service-Oriented Application Design with Effect

Learning Objectives

- Understand the concept of a "service" in Effect
- Gain experience with building and composing services
- Explore the motivation behind the `Layer` type
- Learn best practices for structuring an application

Effect

Introduction to Services

Purpose of a Service

Code to an interface, not an implementation

- Abstracts Functionality
- Useful for Prototyping
- Facilitates Easier Testing
- Composition & Modularity

Service Terminology

Term	Definition
`Service`	An interface that exposes a particular set of functionality
`Tag`	A unique type-level & runtime identifier for a service
`Context`	A container which holds a map of `Tag → Service`

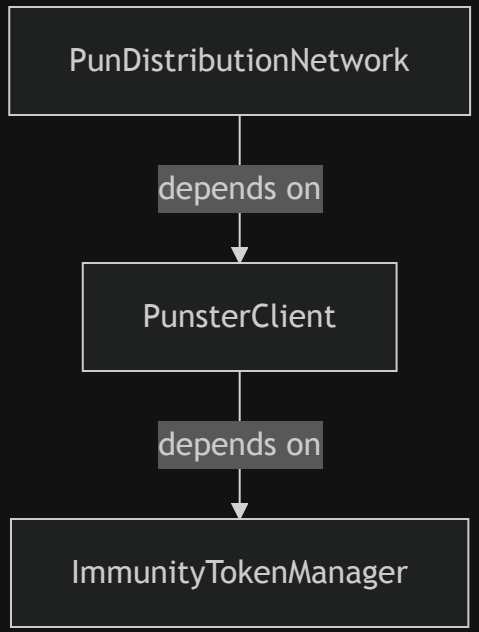
Defining a Service

```ts
import type { Effect } from "effect"
import { Context, Data } from "effect"

// Create a custom error using `Data.TaggedError`. Instances
// of the error will have a `_tag` property of `"CacheMissError"`.
class CacheMissError extends Data.TaggedError("CacheMissError")<{
  readonly message: string
}> { }

// Define the service identifier, shape, and Tag at once
class Cache extends Context.Tag("app/Cache")<Cache, {
  readonly lookup: (key: string) => Effect.Effect<string, CacheMissError>
}>() { }
```

Exercise: Creating a Service

`src/exercises/section-1/001_creating-a-service.ts`

The Pun-ishment Protocol



Exercise: Creating a Service

`src/exercises/section-1/001_creating-a-service.ts`

Create the service definitions for the following:

- `PunDistributionNetwork`
- `PunsterClient`
- `ImmunityTokenManager`

Exercise Recap: Creating a Service

`src/exercises/section-1/001_creating-a-service-solution.ts`

- We imported the `Context` module from `"effect"`
- We used `Context.Tag` to create unique service identifiers

Using a Service

```
ServiceTag.pipe(
  Effect.andThen((serviceImpl) ⇒ ...)
)

// or

Effect.gen(function*() {
  const serviceImpl = yield* ServiceTag
  ...
})
```

Effect

Using a Service

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

class CacheMissError extends Data.TaggedError("CacheMissError")<{
  readonly message: string
}> {}

class Cache extends Context.Tag("app/Cache")<Cache, {
  readonly lookup: (key: string) => Effect.Effect<string, CacheMissError>
}>() {}

//      ┌─── Effect<void, CacheMissError, Cache>
//      ▼
const program = Effect.gen(function*() {
  // Retrieve the Cache service from the Context
  const cache = yield* Cache
  // Use the Cache service
  const value = yield* cache.lookup("my-key")
  console.log(value)
})
```

Exercise: Using a Service

`src/exercises/section-1/002_using-a-service.ts`

Define the `main` Effect:

- Access the requisite services in the program
- Use the service interfaces to implement the business logic

Exercise Recap: Using a Service

`src/exercises/section-1/002_using-a-service-solution.ts`

- We accessed services via their `Tag`s
- We used the service interfaces to implement our business logic
- We observed that we have not *implemented* our services yet
- We observed that services are tracked in the `Requirements` type

Providing a Service

```
effect.pipe(
  // Associate a concrete implementation with its Tag
  Effect.provideService(ServiceTag, serviceImpl)
)

// or

effect.pipe(
  // Associate an Effect that produces a concrete implementation
  // with its Tag
  Effect.provideServiceEffect(ServiceTag, Effect<serviceImpl, ...>)
)
```

Effect

Providing a Service

```ts
class Cache extends Context.Tag("app/Cache")<Cache, {
  readonly lookup: (key: string) => Effect.Effect<string, CacheMissError>
}>() {}

//       ┌─── Effect<void, CacheMissError, Cache>
//       ▼
const program = Effect.gen(function*() {
  const cache = yield* Cache
  const value = yield* cache.lookup("my-key")
  console.log(value)
})

//        ┌─── Effect<void, CacheMissError, never>
//        ▼
const runnable = program.pipe(
  Effect.provideService(Cache, {
    lookup: (key) => Effect.succeed(`${key}-value`)
  })
)

Effect.runPromise(runnable)
// => "my-key-value"
```

Demo: Providing a Service

Single Implementation

`src/demos/section-1/001_providing-a-service-00.ts`

Demo: Providing a Service

Multiple Implementations

`src/demos/section-1/001_providing-a-service-01.ts`

Exercise: Providing a Service

`src/exercises/section-1/003_providing-a-service.ts`

Create a test implementation of the `PunsterClient`:

- Should always return the same `Pun` from `createPun`
- Should always return the same evaluation from `evaluatePun`
- Provide the test implementation to the `main` program

Exercise Recap: Providing a Service

`src/exercises/section-1/003_providing-a-service-solution.ts`

- We created a test implementation of our `PunsterClient`
- We provided the implementation to `main`
- We observed that swapping implementations is trivial

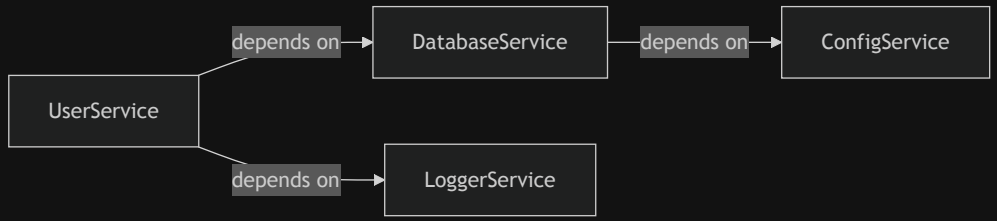
FileSystemCache

This is not testable!

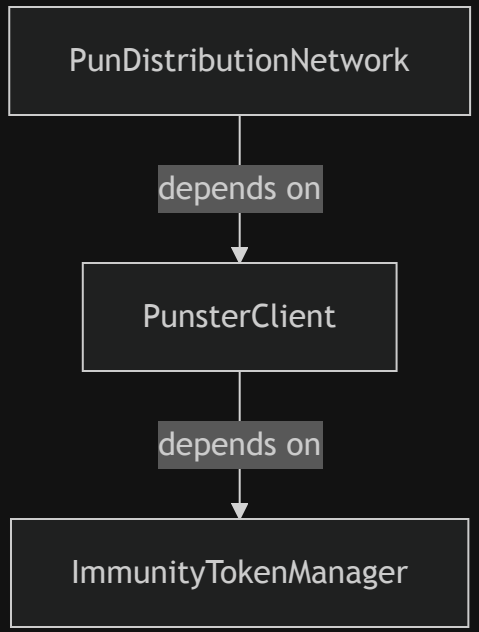
```ts
import { Context, Data, Effect } from "effect"
import * as fs from "node:fs/promises"

// ... <snip> ...

const FileSystemCache = Cache.of({
  lookup: (key) =>
    Effect.tryPromise({
      try: () => fs.readFile(`src/demos/section-1/cache/${key}`, "utf-8"),
      catch: () =>
        new CacheMissError({
          message: `Failed to read file for cache key: "${key}"`
        })
    })
})
```

Services with Dependencies

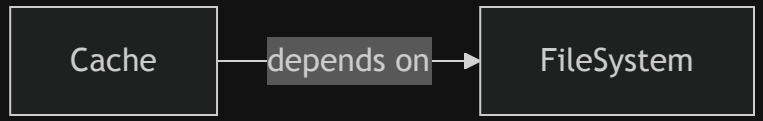
- Services can have dependencies on other services
- Naturally results in a directed acyclic graph of services



Services with Dependencies



Fixing the FileSystemCache



A cache that depends on a file system

Avoid Leaking Requirements

```ts
import type { Effect } from "effect"
import { Context, Data } from "effect"

class FileReadError extends Data.TaggedError("FileReadError")<{
  readonly message: string
}> {}

class FileSystem extends Context.Tag("app/FileSystem")<FileSystem, {
  readonly readFileString: (path: string) => Effect.Effect<string, FileReadError>
}>() {}

class CacheMissError extends Data.TaggedError("CacheMissError")<{
  readonly message: string
}> {}

class Cache extends Context.Tag("app/Cache")<Cache, {
  // ✅ FileSystem is not leaked into the interface
  readonly lookup: (key: string) => Effect.Effect<string, CacheMissError>
}>() {}
```

Effect

Providing Dependent Services

```ts
//       ┌─── Effect<void, CacheMissError, FileSystem> ⛔
//       ▼
const nonRunnable = program.pipe(
  Effect.provideServiceEffect(Cache, makeFileSystemCache)
)

// Providing a FileSystem implementation eliminates
// all remaining service dependencies, making the Effect runnable
//
//       ┌─── Effect<void, CacheMissError> 🎉
//       ▼
const runnable = nonRunnable.pipe(
  Effect.provideService(FileSystem, {
    readFileString: (path) =>
      Effect.tryPromise({
        try: () => fs.readFile(path, "utf-8"),
        catch: () => new FileReadError({ message: `Failed to read file at path "${path}"` })
      })
  })
)

Effect.runPromise(runnable)
```

Going Further

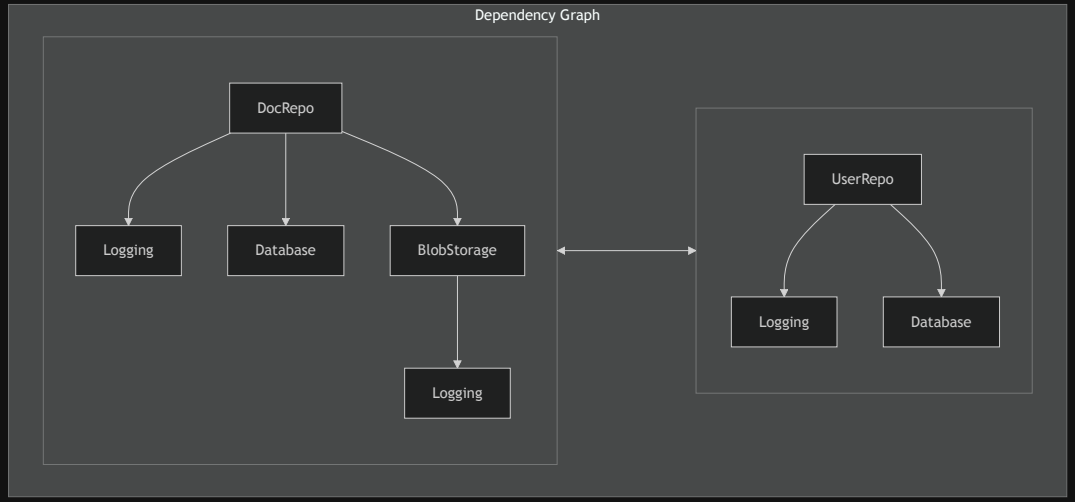
As the number of services grow, thair relational complexity grows

How do we deal with...

- Service composition?
- Singleton services?
- Resource safety?

Effect

Complex Service Relationships



Effect

Introduction to Layers

Issues with Services

- A service may have one or more dependencies
- A service must have dependencies provided in the correct order
- A service might be **resourceful**

Layer as Service Constructor

A data type which represents a constructor for one or more services

- May depend on other services
- May fail to construct a service, producing some error value
- May manage the acquisition / release of resources
- Easily composable with other `` `Layer` `` s
- Are memoized during resolution of the dependency graph

The `Layer` Type

```
        ┌── The service(s) to be created
        │               ┌── The possible error
        │               │      ┌── The required dependencies
        ▼               ▼      ▼
Layer<RequirementsOut, Error, RequirementsIn>
```

Technically...

Creating a Layer

```
// Synchronous layers
Layer.sync(
  ServiceTag,
  // A thunk that produces the concrete service implementation
  () => ServiceShape
) // → Layer<ServiceTag, never, never>

// Asynchronous layers
Layer.effect(
  ServiceTag,
  // An Effect that produces the concrete service implementation
  Effect<ServiceShape, ...>
) // → Layer<ServiceTag, never, never>

// Resourceful layers
Layer.scoped(
  ServiceTag,
  // A scoped Effect that resourcefully produces the concrete
  // service implementation
  Effect<ServiceShape, ..., Scope>
) // → Layer<ServiceTag, never, Scope>
```

Effect

Creating a Layer (Example)

```ts
//          ┌─── Layer<FileSystem, never, never>
//          ▼
const FileSystemLayer = Layer.succeed(FileSystem, {
  readFileString: (path) =>
    Effect.tryPromise({
      try: () => fs.readFile(path, "utf-8"),
      catch: () => new FileReadError({ message: `Failed to read file at path "${path}"` })
    })
})

// Providing the `FileSystemLayer` to the `CacheLayer`
// results in a `Layer<Cache, never, never>`
//
//       ┌─── Layer<Cache, never, never>
//       ▼
const MainLayer = Layer.provide(CacheLayer, FileSystemLayer)

//       ┌─── Effect<void, CacheMissError, never>
//       ▼
const runnable = Effect.provide(program, MainLayer)

Effect.runPromise(runnable)
```

Exercise: Creating a Layer

`src/exercises/section-1/004_creating-a-layer.ts`

Define `Layer`s for each of our services:

- `PunDistributionNetworkLayer`
- `PunsterClientLayer`
- `ImmunityTokenManagerLayer`

Exercise Recap: Creating a Layer

`src/exercises/section-1/004_creating-a-layer-solution.ts`

- We imported `Layer` from the `"effect"` module
- We used `Layer.effect` for non-resourceful services
- We used `Layer` combinators to compose `Layer`s together
- We have a lingering `Scope` in our requirements

Effect

Simplifying Service Definitions

```
      )
    }
    return { lookup } as const
  }),
  // Provide service dependencies (optional)
  dependencies: [FileSystemLayer]
}) {}

// This layer is automatically generated by `Effect.Service` and
// will build the `Cache` service
//
//       ┌─── Layer<Cache, never, never>
//       ▼
Cache.Default

// This layer is automatically generated by `Effect.Service` and
// will build the `Cache` service without any dependencies provided
// (only generated if you specify `dependencies`)
//
//              ┌─── Layer<Cache, never, FileSystem>
//              ▼
Cache.DefaultWithoutDependencies
```

Effect

Simplifying Service Definitions (Example)

```
    return { lookup } as const
  }),
  dependencies: [FileSystem.Default]
}) {}

//       ┌─── Layer<Cache, never, never>
//       ▼
Cache.Default

//       ┌─── Effect<void, CacheMissError, Cache>
//       ▼
const program = Effect.gen(function*() {
  const cache = yield* Cache
  const data = yield* cache.lookup("my-key")
  console.log(data)
})

//       ┌─── Effect<void, CacheMissError, never>
//       ▼
const runnable = Effect.provide(program, Cache.Default)

Effect.runPromise(runnable)
```

Exercise: Simplifying Service Definitions

`src/exercises/section-1/005_simplifying-service-definitions.ts`

Re-define our services using `Effect.Service`:

- `PunDistributionNetwork`
- `PunsterClient`
- `ImmunityTokenManager`

Exercise Recap: Simplifying Service Definitions

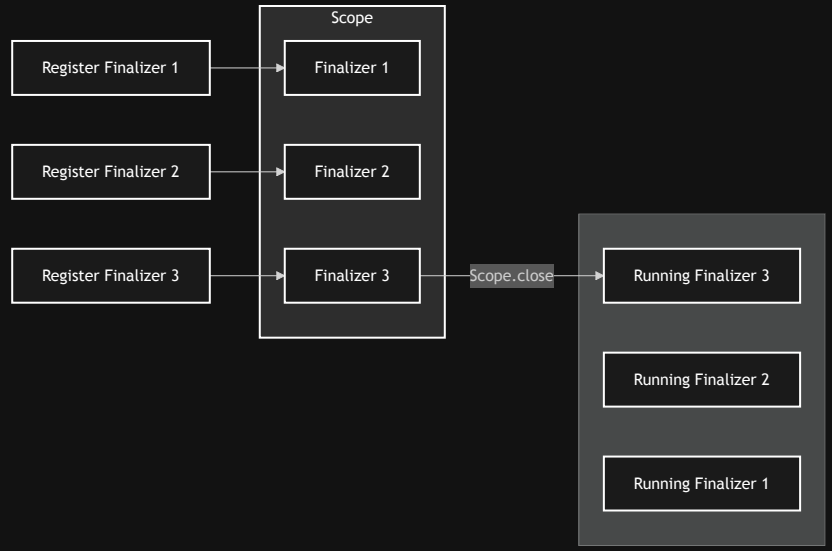
`src/exercises/section-1/005_simplifying-service-definitions-solution.ts`

- We used `Effect.Service` to define both a `Tag` and `Layer`
- We used the `dependencies` to locally provide dependencies
- We still have that lingering scope in the requirements

Effect

Resourceful Layers

A Quick Aside on Scope



Effect

Scoped Effects (Example)

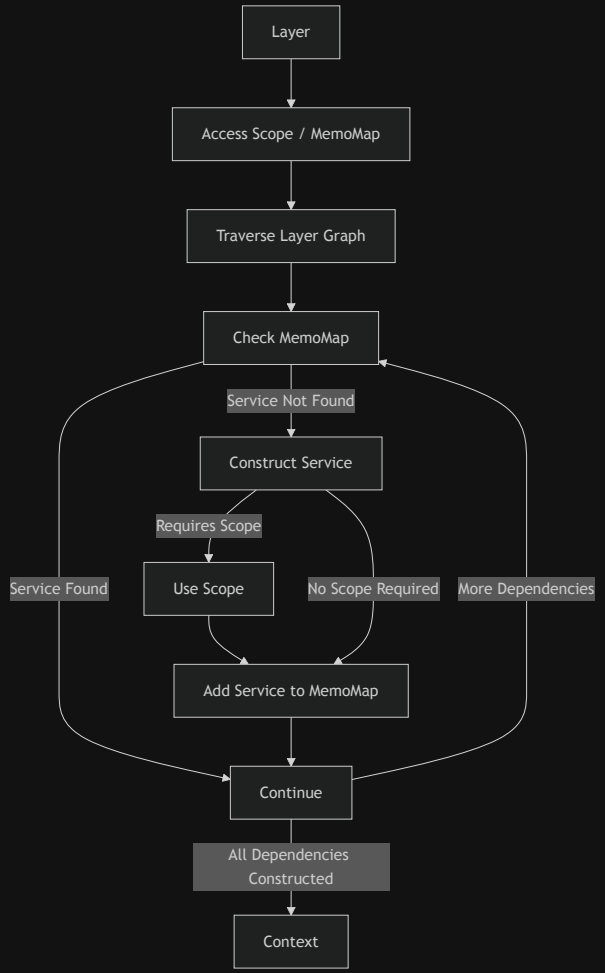
```
27
28  // Providing a `Scope` to the program will erase the `Scope`
29  // from the requirements and extend that `Scope` to all
30  // resourceful operations in the program. Remember, the
31  // finalizers will not be run until the `Scope` is closed.
32  //
33  // Here, `Effect.scoped` creates a `Scope`, provides it to
34  // the program, and closes the `Scope` when the program exits.
35  //
36  //       ┌─── Effect<void, never, never>
37  //       ▼
38  const runnable = Effect.scoped(program)
39
40  Effect.runPromise(runnable)
41  /*
42  Output:
43  Acquiring 1
44  Acquiring 2
45  Acquiring 3
46  Releasing 3
47  Releasing 2
48  Releasing 1
49  */
```

Effect

Using a `Scope`

```
9     const program = Effect.acquireRelease(
10      // Acquire the resource
11      Effect.log("Acquiring resource...").pipe(
12        Effect.as("Hello, World!")
13      ),
14      // Define a finalizer for the resource. Finalizers have
15      // access to the `Exit` value provided when the `Scope`
16      // is closed (via `Scope.close`), which is usually the
17      // `Exit` value of scoped Effect that was executed.
18      //
19      // ┌─── string
20      // │         ┌─── Exit<unknown, unknown>
21      // ▼         ▼
22      (resource, exit) => Effect.log("Releasing resource...")
23    )
24
25    yield* Scope.extend(program, scope)
26
27    const exit = yield* Effect.exit(program)
28
29    yield* Scope.close(scope, exit)
30  })
```

Effect


Layer
Access Scope / MemoMap
Traverse Layer Graph
Check MemoMap
Service Not Found
Construct Service
Requires Scope
Service Found
Use Scope
No Scope Required
More Dependencies
Add Service to MemoMap
Continue
All Dependencies Constructed
Context

Effect

Providing a Layer (Example)

```
37    const layerBuilder = Layer.build(MyServiceLayer)
38
39    // Build a `Context` from the `Layer` containing all
40    // the services in the `Layer`'s dependency graph.
41    //
42    //       ┌─── Context<MyService>
43    //       ▼
44    const context = yield* Scope.extend(layerBuilder, scope)
45
46    // Provide the built `Context` to the `program` to
47    // satisfy all of `program`'s requirements.
48    //
49    //        ┌─── Effect<void, never, never>
50    //        ▼
51    const runnable = Effect.provide(program, context)
52
53    // Run the program and obtain its exit value
54    const exit = yield* Effect.exit(runnable)
55
56    // Close the scope
57    yield* Scope.close(scope, exit)
58  })
```

Effect

Resourceful Layers (Example)

```
        )
      }

      return { cacheDir, lookup } as const
    }),
    dependencies: [FileSystem.Default]
  }) {}

  //      ┌─── Effect<void, CacheMissError, Cache | FileSystem>
  //      ▼
  const program = Effect.gen(function*() {
    const cache = yield* Cache
    const fs = yield* FileSystem
    yield* fs.writeFileString(`${cache.cacheDir}/key1`, "value1")
    const value = yield* cache.lookup("key1")
    console.log(value)
  })

  program.pipe(
    Effect.provide([Cache.Default, FileSystem.Default]),
    Effect.runPromise
  )
```

Exercise: Resourceful Layers

`src/exercises/section-1/006_resourceful-layers.ts`

Remove the `Scope` requirement from our final `Layer`:

- Resourceful services should cleanup when the program ends

Exercise Recap: Resourceful Layers

`src/exercises/section-1/006_resourceful-layers-solution.ts`

- We used `Layer.scoped` to control the liftime of resources acquired during service construction
- We locally eliminated requirements for each of our services
- We created a `MainLayer` which combines all of our services
- We provided a `NodeHttpClient` to the `MainLayer` to satisfy all requirements

Layer Composition

There are two primary methods for `Layer` composition:

- **Merging** - merges the inputs and outputs of two layers together
- **Providing** - provides the outputs of one layer as inputs to another

Merging Layers

```ts
import { Layer } from "effect"

declare const layer1: Layer.Layer<"Out1", never, "In1">
declare const layer2: Layer.Layer<"Out2", never, "In2">

//      ┌─── Layer<"Out1" | "Out2", never, "In1" | "In2">
//      ▼
const merged = Layer.merge(layer1, layer2)
```

Providing Layers

```ts
import { Layer } from "effect"

declare const layer1: Layer.Layer<"Out1", never, "Requirement">
declare const layer2: Layer.Layer<"Requirement", never, "In2">

//        ┌─── Layer<"Out1", never, "In2">
//        ▼
const provided = Layer.provide(layer1, layer2)
```

Providing & Merging Layers

```ts
import { Layer } from "effect"

declare const layer1: Layer.Layer<"Out1", never, "Requirement">
declare const layer2: Layer.Layer<"Requirement", never, "In2">

//          ┌─── Layer<"Out1" | "Requirement", never, "In2">
//          ▼
const providedAndMerged = Layer.provideMerge(layer1, layer2)
```

Exercise: Layer Composition

`src/exercises/section-1/007_layer-composition.ts`

Remove the `Scope` dependency from our final `Layer`:

- Resourceful services should cleanup when the program ends

Exercise Recap: Layer Composition

`src/exercises/section-1/007_layer-composition-solution.ts`

- We practiced `Layer` composition using the following methods:
  - `Layer.provide`
  - `Layer.merge`
  - `Layer.provideMerge`
- We observed why local elimination of requirements is recommended

Designing around Layers

- Identify key subsystems within an application
- Decompose these subsystems into services
- Build up a set of top-level layers to provide

Layers - Best Practices

- Use `Effect.Service` wherever possible
- Locally provide service dependencies (if possible)
- Avoid multiple calls to `Effect.provide`
- Remember that `Layer`s are memoized by *reference*

Exercise: Running the Pun-ishment Protocol

`src/exercises/section-1/008_running-the-application.ts`

Run the Pun-ishment Protocol application!

- You can use the following command to run the file

```
pnpm exercise ./src/exercises/section-1/008_running-the-application.ts
```

Exercise Recap: Running the Punishment Protocol

`src/exercises/section-1/008_running-the-application.ts`

- We combined our `Layer`s into a `MainLayer`
- We used `Effect.provide` to provide our `Layer` to the `main` program
- We ran our program and observed the output

Section Two

Incremental Adoption of Effect

Learning objectives

- Learn how to integrate Effect into an existing codebase
- Understand strategies for wrapping existing business logic into Effect layers
- Gain experience in incrementally adopting Effect
- Explore interoperability with non-Effect code

So, you want to adopt Effect?

- But you are already using other frameworks
- You are using libraries with Promise-based APIs
- Existing code isn't written holistically
  - Error handling
  - Resource management
  - Interruption
  - Observability

Holistic thinking

When adopting Effect, you start thinking about things that you might have overlooked before.

- What kind of errors can occur?
- Are resources being alloacted here? And how should I release them?
- How do I abort expensive computations?
- How do I monitor the execution of this code?

Effect

Wrapping Promise-based libraries

The `use` pattern

A common approach to wrapping Promise-based libraries with Effect is to create an Effect.Service that exposes an `use` method.

```
interface SomeApi {}

declare const use: <A>(f: (api: SomeApi) => Promise<A>) => Effect<A>
```

Effect

Demo: Wrapping OpenAI

`src/demos/section-2/openai-00.ts`

OpenAI demo recap

- We used Effect.Service with the `use` pattern to wrap the OpenAI library
- We used the `Config` module to retrieve the client configuration
- We used `Schema.TaggedError` to wrap errors (you could also use `Data.TaggedError`)
- We considered interruption by passing an `AbortSignal`
- We used `Effect.fn` to add tracing to our `use` method
- There was no "resources" that needed to be managed

Questions?

Exercise: Wrap a sqlite client

`src/exercises/section-2/sqlite-01.ts`

Wrapping paginated APIs

A lot of APIs return paginated data. How do we wrap them with Effect?

- Streams!
  - `` `import { Stream } from "effect"` ``
- Stream.paginate*

Stream.paginateChunkEffect

Allows you to continuously fetch data using some kind of cursor.

```
export const paginateChunkEffect: <S, A, E, R>(
  s: S,
  f: (s: S) => Effect.Effect<readonly [Chunk.Chunk<A>, Option.Option<S>], E, R>,
) => Stream<A, E, R>
```

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

Stream.paginateChunkEffect(1, (page) =>
  fetchPage(page).pipe(
    Effect.map((items) => [Chunk.unsafeFromArray(items), Option.some(page + 1)]),
  ),
)
```

Demo: OpenAI pagination

`src/demos/section-2/openai-paginate-00.ts`

OpenAI pagination recap

- We used `` `Stream.paginateChunkEffect` `` to continuously fetch data
- We tracked the cursor using OpenAI's Page api
- We used `` `Effect.fn` `` & `` `Stream.withSpan` `` to add tracing to our stream

Questions?

Wrapping specialized APIs

- Some libraries have more specific APIs, like streaming completions from OpenAI
- This may require adding additional service methods to make usage more ergonomic

Demo: OpenAI streaming completions

`src/demos/section-2/openai-completions-00.ts`

OpenAI completions example recap

- For commonly used APIs, it may be beneficial to create seperate service methods to improve ergonomics
- There is often Effect API's you can use to wrap common JavaScript data types. I.e. we used `` `Stream.fromAsyncIterable` `` to wrap the OpenAI stream.
- Creating ergonomic APIs can require some reverse engineering effort

Questions?

Exercise: Create sqlite stream API

`src/exercises/section-2/sqlite-02.ts`

Wrapping multi-shot APIs

In some scenarios, you need to wrap an API that invokes a callback multiple times, such as request handlers or event listeners.

- You often want to access your Effect services in these callbacks
- You need to ensure that fibers are properly managed, to prevent leaks
    - Fibers represent a running Effect computation

Multi-shot integration strategies

1. Directly fork a fiber in the callback, and subscribe to the result
2. Indirectly fork a fiber, by adding requests to a queue and processing them in a worker. If the callback requires a response, send back a signal using a `` `Deferred` ``.
3. Convert the callback into a stream (if the callback doesn't require a response)

Stream.async APIs

If you don’t need to return a value to the callback, you can convert the multi-shot callback into a Stream, using the `Stream.async` family of functions.

This is useful for event based APIs.

- `Stream.async` & `Stream.asyncScoped` - for when the callback supports back-pressure
- `Stream.asyncPush` - for when the backing API doesn’t support back-pressure

Demo: Event listeners

`src/demo/section-2/events-00.ts`

Effect provided utilities

There are several utilities provided by Effect for wrapping JavaScript data sources:

- `Stream.fromEventListener` - for wrapping event listeners
- `Stream.fromAsyncIterable` - for wrapping async iterables
- `Stream.fromReadableStream` - for wrapping web readable streams
- `NodeStream.fromReadable` - for wrapping Node.js readable streams
  - `import { NodeStream } from "@effect/platform-node"`
- `NodeSink.fromWritable` - for wrapping Node.js writable streams
  - `import { NodeSink } from "@effect/platform-node"`

Forking fibers directly

There are several options for running Effect's:

1. Use `Effect.runFork`, `Effect.runPromise` etc.
2. Use `Effect.runtime` to access the current runtime for running Effect's, which is useful if you need to access services
3. Use the `FiberSet` module to manage fibers, which adds life-cycle management

FiberHandle / FiberSet / FiberMap

When managing one or many fibers, the `` `Fiber{Handle,Set,Map}` `` modules can be used to ensure that the lifecycle of the fibers are managed correctly.

- `` `FiberHandle` `` - for managing a single fiber
  - Useful for managing a server that needs to be started and stopped
- `` `FiberSet` `` - for managing multiple fibers without any identity
  - Useful for managing request handlers
- `` `FiberMap` `` - for managing multiple fibers with keys / identity
  - Useful for managing a well-known set of fibers, like a group of background tasks indexed by a key

Demo: Wrapping express

`src/demos/section-2/express-00.ts`

Express demo recap

- We used `FiberSet` to run the request handlers
- We used `Effect.acquireRelease` to ensure the server is properly shut down
- We used our Effect services by accessing them outside of the request handlers

There are some issues to solve

- How do we compose different parts of a large application together?
- Maybe we want to test different parts of the application in isolation?
- Improve error handling and make it more ergonomic

Demo: Wrapping express with Layer

`src/demos/section-2/express-layer-00.ts`

Express with Layer recap

- We used `Layer` to compose different parts of the application together
- We added a `addRoute` helper to ensure request handlers consider:
  - Error handling
  - Interruption
  - Observability

Exercise: Migrate express app to Effect

`src/exercises/section-2/express/main.ts`

Effect in the frontend

When using Effect in frontend frameworks, it requires a different approach compared to the backend, as you often don’t control the "main" entry-point.

Effect frontend strategies

- Use `ManagedRuntime` to integrate Effect services into components
  - Use React's context API to provide the runtime to your components
- Experimental: `@effect-rx/rx` package
  - Provides a jotai-like API for integrating Effect with frameworks like React
  - Integration packages: `@effect-rx/rx-react` & `@effect-rx/rx-vue`

`ManagedRuntime`

Create a runtime that can execute Effect's from a Layer

```ts
import { Effect, ManagedRuntime } from "effect"
import { OpenAi } from "./services.js"

// OpenAi.Default: Layer<OpenAi>

const runtime = ManagedRuntime.make(OpenAi.Default)

declare const effect: Effect.Effect<void, never, OpenAi>

runtime.runPromise(effect)
```

`Layer.MemoMap`

- A "black box" data type that can memoize the result of building a Layer.
- Relatively low-level, but can be used to ensure the same Layer is only built once across your application.

```
import { Effect, Layer, ManagedRuntime } from "effect"
import { OpenAi } from "./services.js"

const memoMap = Effect.runSync(Layer.makeMemoMap)
const runtimeA = ManagedRuntime.make(OpenAi.Default, memoMap)
const runtimeB = ManagedRuntime.make(OpenAi.Default, memoMap)
```

Effect

Demo: Using Effect in React

`src/demos/section-2/react`

React demo recap

- We used `ManagedRuntime` to consume Effect services in React components
  - Wrapped with React's context API and `useEffect` to manage the runtime lifecycle
- We used a global `MemoMap` to ensure that the same Layer is only built once if used in multiple `ManagedRuntime` instances
- We passed the `AbortSignal` from `@tanstack/react-query` to the `runPromise` call, to integrate with Effect's interruption model
- We integrated `Stream` with React's `useEffect`

Questions?

Exercise: Migrate React Pokemon app to Effect

`src/exercises/section-2/react`

- Run it with `pnpm vite src/exercises/section-2/react`