ScalaJobs

# Resource Management Made Easy: Introducing Resource from Cats-Effect

julien@scalajobs.com

# Julien Truffaut

ScalaJobs

Backend Scala developer for 10+ years

Teach Scala and functional programming at fp-tower.com

Co-founder of scalajobs.com

# Resource

ScalaJobs

1. Initialisation
2. Usage
3. Finalisation

# Example 1: File

ScalaJobs

1. Open a file
2. Read/Write data into the file
3. Close the file

# Example 2: HTTP Server

ScalaJobs

1. Configure and start an HTTP server
2. Serve HTTP requests
3. Close the server

# Example 3: Integration Tests

ScalaJobs

1. Create tables and setup test data
2. Run the tests
3. Drop or truncate the tables

# Example 4: Queue

ScalaJobs

1. Pull an event from a queue
2. Process the event
3. On error, requeue the event

ScalaJobs

How to ensure the finalisation logic is executed?

in all circumstances

ScalaJobs

# Solutions

# Shutdown Hook

ScalaJobs

```
object Main extends App {

  sys.addShutdownHook {
    println("Application is shutting down")
  }

  println("Application is ready")

}
```

# Shutdown Hook

ScalaJobs

```
object Main extends App {

  sys.addShutdownHook {
    println("Application is shutting down")
  }

  println("Application is ready")

}
```

# Shutdown Hook

ScalaJobs

```
object Main extends App {

  sys.addShutdownHook {
    println("Application is shutting down")
  }

  println("Application is ready")

}
```

# Shutdown Hook

ScalaJobs

```
object Main extends App {

  sys.addShutdownHook {
    println("Application is shutting down")
  }

  println("Application is ready")

}
// Application is ready
// Application is shutting down
```

# Shutdown Hook

ScalaJobs

```
object Main extends App {

  val httpServer = ... // configure

  sys.addShutdownHook {
    httpServer.close()
  }

  httpServer.start()

}
```

# Before and After All

ScalaJobs

```
import org.scalatest.BeforeAndAfterAll
import org.scalatest.funsuite.AnyFunSuite

class UserRepositoryTest
  extends AnyFunSuite
  with    BeforeAndAfterAll {

  override def beforeAll(): Unit = {
    // create user table
  }

  override def afterAll(): Unit = {
    // drop user table
  }

  test("create a new user") {
    userRepo.create(User("1234", "John", "Doe"))

    assert(userRepo.get("1234").isDefined)
  }

}
```

# Before and After All

ScalaJobs

```
import org.scalatest.BeforeAndAfterAll
import org.scalatest.funsuite.AnyFunSuite

class UserRepositoryTest
  extends AnyFunSuite
  with    BeforeAndAfterAll {

  override def beforeAll(): Unit = {
    // create user table
  }

  override def afterAll(): Unit = {
    // drop user table
  }

  test("create a new user") {
    userRepo.create(User("1234", "John", "Doe"))

    assert(userRepo.get("1234").isDefined)
  }

}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: IOException => ...
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
try {
  val reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} finally {
  reader.close() // ❌ reader isn't visible
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt")) // FileNotFoundException
  reader.readLine()
} finally {
  reader.close()
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt")) // FileNotFoundException
  reader.readLine()
} finally {
  reader.close() // NullPointerException
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt")) // FileNotFoundException
  reader.readLine()
} finally {
  reader.close() // NullPointerException
}
// 1) FileNotFoundException
// 2) NullPointerException
// 3) Something else?
```

# Try-Catch-Finally

ScalaJobs

```
var reader: BufferedReader = null

try {
  reader = new BufferedReader(new FileReader("text.txt")) // FileNotFoundException
  reader.readLine()
} finally {
  reader.close() // NullPointerException
}
// 1) FileNotFoundException
// 2) NullPointerException
// 3) Something else?
```

# Try-Catch-Finally

ScalaJobs

```
var reader    : BufferedReader = null
var errorInTry: Throwable  = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: Throwable =>
   errorInTry = e
   throw e
} finally {
  try {
    reader.close()
  } catch {
    case errorInFinally: Throwable =>
      if(errorInTry != null) {
        errorInTry.addSuppressed(errorInFinally)
        throw errorInTry
      } else
        throw errorInFinally
  }
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader    : BufferedReader = null
var errorInTry: Throwable  = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: Throwable =>
   errorInTry = e
   throw e
} finally {
  try {
    reader.close()
  } catch {
    case errorInFinally: Throwable =>
      if(errorInTry != null) {
        errorInTry.addSuppressed(errorInFinally)
        throw errorInTry
      } else
        throw errorInFinally
  }
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader    : BufferedReader = null
var errorInTry: Throwable  = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: Throwable =>
   errorInTry = e
   throw e
} finally {
  try {
    reader.close()
  } catch {
    case errorInFinally: Throwable =>
      if(errorInTry != null) {
        errorInTry.addSuppressed(errorInFinally)
        throw errorInTry
      } else
        throw errorInFinally
  }
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader    : BufferedReader = null
var errorInTry: Throwable  = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: Throwable =>
   errorInTry = e
   throw e
} finally {
  try {
    reader.close()
  } catch {
    case errorInFinally: Throwable =>
      if(errorInTry != null) {
        errorInTry.addSuppressed(errorInFinally)
        throw errorInTry`
      } else
        throw errorInFinally
  }
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader    : BufferedReader = null
var errorInTry: Throwable  = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: Throwable =>
   errorInTry = e
   throw e
} finally {
  try {
    reader.close()
  } catch {
    case errorInFinally: Throwable =>
      if(errorInTry != null) {
        errorInTry.addSuppressed(errorInFinally)
        throw errorInTry
      } else
        throw errorInFinally
  }
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader    : BufferedReader = null
var errorInTry: Throwable  = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: Throwable =>
   errorInTry = e
   throw e
} finally {
  try {
    reader.close()
  } catch {
    case errorInFinally: Throwable =>
      if(errorInTry != null) {
        errorInTry.addSuppressed(errorInFinally)
        throw errorInTry
      } else
        throw errorInFinally
  }
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader    : BufferedReader = null
var errorInTry: Throwable  = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: Throwable =>
   errorInTry = e
   throw e
} finally {
  try {
    reader.close()
  } catch {
    case errorInFinally: Throwable =>
      if(errorInTry != null) {
        errorInTry.addSuppressed(errorInFinally)
        throw errorInTry
      } else
        throw errorInFinally
  }
}
```

# Try-Catch-Finally

ScalaJobs

```
var reader    : BufferedReader = null
var errorInTry: Throwable  = null

try {
  reader = new BufferedReader(new FileReader("text.txt"))
  reader.readLine()
} catch {
  case e: Throwable =>
   errorInTry = e
   throw e
} finally {
  try {
    reader.close()
  } catch {
    case errorInFinally: Throwable =>
      if(errorInTry != null) {
        errorInTry.addSuppressed(errorInFinally)
        throw errorInTry
      } else
        throw errorInFinally
  }
}
```

# Using

from package scala.util (from 2.13)

# Using

ScalaJobs

```
import scala.util.Using

Using {
  new BufferedReader(new FileReader("text.txt"))
} { reader =>
  reader.readLine()
}
```

# Using

ScalaJobs

```scala
import scala.util.Using

Using {
  new BufferedReader(new FileReader("text.txt"))
} { reader =>
  reader.readLine()
}
```

# Using

ScalaJobs

```scala
import scala.util.Using

Using {
  new BufferedReader(new FileReader("text.txt"))
} { reader =>
  reader.readLine()
}
```

# Using

ScalaJobs

```
import scala.util.Using

Using {
  new BufferedReader(new FileReader("text.txt"))
} { reader =>
  reader.readLine()
}
```

Close resource automatically

Handle multiple errors

# Using returns a Try

ScalaJobs

```
import scala.util.{Using, Try}

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
// res: Try[String] = ...
```

# Using returns a Try

ScalaJobs

```scala
import scala.util.{Using, Try}

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
// res: Try[String] = ...
```

# Using returns a Try

ScalaJobs

```
import scala.util.{Using, Try}

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
// res: Try[String] = Success("Hello World")
```

# Using returns a Try

ScalaJobs

```scala
import scala.util.{Using, Try}

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
// res: Try[String] = Failure(java.io.FileNotFoundException: test.txt)
```

# Using with multiple resources

ScalaJobs

```scala
import scala.util.Using

Using.Manager { register =>
  val fileToRead  = register(new BufferedReader(new FileReader("file1.txt")))
  val fileToWrite = register(new BufferedWriter(new FileWriter("file1-copy.txt")))

  Iterator
    .continually(reader.readLine())
    .takeWhile(_ != null)
    .foreach { line =>
      writer.write(line)
      writer.newLine()
    }

}
```

# Using with multiple resources

ScalaJobs

```scala
import scala.util.Using

Using.Manager { register =>
  val fileToRead  = register(new BufferedReader(new FileReader("file1.txt")))
  val fileToWrite = register(new BufferedWriter(new FileWriter("file1-copy.txt")))

  Iterator
    .continually(reader.readLine())
    .takeWhile(_ != null)
    .foreach { line =>
      writer.write(line)
      writer.newLine()
    }

}
```

# Using with multiple resources

ScalaJobs

```scala
import scala.util.Using

Using.Manager { register =>
  val fileToRead  = register(new BufferedReader(new FileReader("file1.txt")))
  val fileToWrite = register(new BufferedWriter(new FileWriter("file1-copy.txt")))

  Iterator
    .continually(reader.readLine())
    .takeWhile(_ != null)
    .foreach { line =>
      writer.write(line)
      writer.newLine()
    }

}
```

# Using with multiple resources

ScalaJobs

```scala
import scala.util.Using

Using.Manager { register =>
  val fileToRead  = register(new BufferedReader(new FileReader("file1.txt")))
  val fileToWrite = register(new BufferedWriter(new FileWriter("file1-copy.txt")))

  Iterator
    .continually(reader.readLine())
    .takeWhile(_ != null)
    .foreach { line =>
      writer.write(line)
      writer.newLine()
    }

}
```

# Using with multiple resources

ScalaJobs

```scala
import scala.util.Using

Using.Manager { register =>
  val fileToRead  = register(new BufferedReader(new FileReader("file1.txt")))
  val fileToWrite = register(new BufferedWriter(new FileWriter("file1-copy.txt")))

  Iterator
    .continually(reader.readLine())
    .takeWhile(_ != null)
    .foreach { line =>
      writer.write(line)
      writer.newLine()
    }

}
```

Close resources in reverse initialisation order

# Using under the hood

ScalaJobs

```
import scala.util.Using

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
```

```
object Using {
  def apply[R: Releasable, A](initialise: => R)(use: R => A): Try[A]
}
```

# Using under the hood

ScalaJobs

```scala
import scala.util.Using

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
```

```scala
object Using {
  def apply[R: Releasable, A](initialise: => R)(use: R => A): Try[A]
}
```

# Using under the hood

ScalaJobs

```
import scala.util.Using

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
```

```
object Using {
  def apply[R: Releasable, A](initialise: => R)(use: R => A): Try[A]
}
```

# Using under the hood

ScalaJobs

```
import scala.util.Using

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
```

```
object Using {
  def apply[R: Releasable, A](initialise: => R)(use: R => A): Try[A]
}
```

# Using under the hood

ScalaJobs

```scala
import scala.util.Using

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
```

```scala
object Using {
  def apply[R: Releasable, A](initialise: => R)(use: R => A): Try[A]
}

trait Releasable[-R] {
  def release(initialise: R): Unit // implemented for all AutoCloseable
}
```

# Using under the hood

ScalaJobs

```
import scala.util.Using

Using {
  new BufferedReader(new FileReader("test.txt"))
} { reader =>
  reader.readLine()
}
```

```
object Using {
  def apply[R: Releasable, A](initialise: => R)(use: R => A): Try[A]
}

trait Releasable[-R] {
  def release(initialise: R): Unit // implemented for all AutoCloseable
}
```

# Using custom finalizer

ScalaJobs

```
Using {
  val database = ...
  database.execute(sql"""
    CREATE TABLE user(
      id   TEXT PRIMARY KEY,
      name TEXT NOT NULL,
      dob  DATE
    )""")
  database
} { database =>

  test("create new user") { ... }

  test("delete user") { ... }

} { using () =>
  database.execute(sql"DROP TABLE user")
  database.close()
}
```

# Using custom finalizer

ScalaJobs

```
Using {
  val database = ...
  database.execute(sql"""
    CREATE TABLE user(
      id   TEXT PRIMARY KEY,
      name TEXT NOT NULL,
      dob  DATE
    )""")
  database
} { database =>

  test("create new user") { ... }

  test("delete user") { ... }

} { using () =>
  database.execute(sql"DROP TABLE user")
  database.close()
}
```

# Using custom finalizer

ScalaJobs

```
Using {
  val database = ...
  database.execute(sql"""
    CREATE TABLE user(
      id   TEXT PRIMARY KEY,
      name TEXT NOT NULL,
      dob  DATE
    )""")
  database
} { database =>

  test("create new user") { ... }

  test("delete user") { ... }

} { using () =>
  database.execute(sql"DROP TABLE user")
  database.close()
}
```

# Using custom finalizer

ScalaJobs

```
Using {
  val database = ...
  database.execute(sql"""
    CREATE TABLE user(
      id   TEXT PRIMARY KEY,
      name TEXT NOT NULL,
      dob  DATE
    )""")
  database
} { database =>

  test("create new user") { ... }

  test("delete user") { ... }

} { using () =>
  database.execute(sql"DROP TABLE user")
  database.close()
}
```

# Using custom finalizer

ScalaJobs

```
Using {
  val database = ...
  database.execute(sql"""
    CREATE TABLE user(
      id   TEXT PRIMARY KEY,
      name TEXT NOT NULL,
      dob  DATE
    )""")
  database
} { database =>

  test("create new user") { ... }

  test("delete user") { ... }

} { using () =>
  database.execute(sql"DROP TABLE user")
  database.close()
}
```

Only for Scala 3

# Using limitations

ScalaJobs

## 1. Finalizer doesn't know if there was an error

```
trait Releasable[-R] {
  def release(initialise: R): Unit
}
```

# Using limitations

ScalaJobs

## 1. Finalizer doesn't know if there was an error

```
trait Releasable[-R] {
  def release(initialise: R): Unit
}
```

# Using limitations

ScalaJobs

1. Finalizer doesn't know if there was an error

2. Only works with synchronous logic

```
Using {
  HttpClient.create(...)
} { client =>
  client.get("https://www.githubstatus.com/api/v2/summary.json")
}
// res: Future[Json] = ...
```

# Using limitations

ScalaJobs

1. Finalizer doesn't know if there was an error

2. Only works with synchronous logic

```
Using {
  HttpClient.create(...)
} { client =>
  client.get("https://www.githubstatus.com/api/v2/summary.json")
}
// res: Future[Json] = ...
```

# Using limitations

ScalaJobs

1. Finalizer doesn't know if there was an error

2. Only works with synchronous logic

```
Using {
  HttpClient.create(...)
} { client =>
  client.get("https://www.githubstatus.com/api/v2/summary.json")
}
// res: Future[Json] = ...
```

# Resource

```
"org.typelevel" %% "cats-effect" % "3.5.1"
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

```
import cats.effect.IO

fileToRead("test.txt")
  .use(reader =>
     IO(reader.readLine())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

```
import cats.effect.IO

fileToRead("test.txt")
  .use(reader =>
     IO(reader.readLine())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

```
import cats.effect.IO

fileToRead("test.txt")
  .use(reader =>
     IO(reader.readLine())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

```
import cats.effect.IO

fileToRead("test.txt")
  .use(reader =>
     IO(reader.readLine())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

```
import cats.effect.IO

fileToRead("test.txt")
  .use(reader =>
     IO(reader.readLine())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

```
import cats.effect.IO

fileToRead("test.txt")
  .use(reader =>
     IO(reader.readLine())
  )
```

## 1. Support Sync and Async

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

```
import cats.effect.IO

fileToRead("test.txt")
  .use(reader =>
     IO(reader.readLine())
  )
```

1. Support Sync and Async
2. Handle errors

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  ...
```

```
import cats.effect.IO

fileToRead("test.txt")
  .use(reader =>
     IO(reader.readLine())
  )
```

1. Support Sync and Async
2. Handle errors
3. Handle cancellation

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  Resource.make(
    IO(new BufferedReader(new FileReader(path)))
  )(reader =>
    IO(reader.close())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  Resource.make(
    IO(new BufferedReader(new FileReader(path)))
  )(reader =>
    IO(reader.close())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  Resource.make(
    IO(new BufferedReader(new FileReader(path)))
  )(reader =>
    IO(reader.close())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  Resource.make(
    IO(new BufferedReader(new FileReader(path)))
  )(reader =>
    IO(reader.close())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  Resource.make(
    IO(new BufferedReader(new FileReader(path)))
  )(reader =>
    IO(reader.close())
  )
```

# Resource

ScalaJobs

```
import cats.effect.kernel.Resource

def fileToRead(path: String): Resource[IO, BufferedReader] =
  Resource.fromAutoCloseable(
    IO(new BufferedReader(new FileReader(path)))
  )
```

## Resource

```
Resource
  .fromAutoCloseable(IO(new FileWriter("test.txt")))
  .use(writer =>
    IO(writer.write("Hello World"))
  )
```

## Using


```
Using {
  new FileWriter("test.txt")
} { writer =>
  writer.write("Hello World")
}
```

# Working with multiple resources

ScalaJobs

```
def  fileToRead(path: String): Resource[IO, BufferedReader] = ...
def fileToWrite(path: String): Resource[IO, BufferedWriter] = ...
```

# Working with multiple resources

ScalaJobs

```
def  fileToRead(path: String): Resource[IO, BufferedReader] = ...
def fileToWrite(path: String): Resource[IO, BufferedWriter] = ...
```

# Working with multiple resources

ScalaJobs

```
def  fileToRead(path: String): Resource[IO, BufferedReader] = ...
def fileToWrite(path: String): Resource[IO, BufferedWriter] = ...

for {
  reader <- fileToRead("file1.txt"),
  writer <- fileToWrite("file1-copy.txt")
} yield (reader, writer)
```

# Working with multiple resources

ScalaJobs

```
def  fileToRead(path: String): Resource[IO, BufferedReader] = ...
def fileToWrite(path: String): Resource[IO, BufferedWriter] = ...

for {
  reader <- fileToRead("file1.txt"),
  writer <- fileToWrite("file1-copy.txt")
} yield (reader, writer)
// res: Resource[IO, (BufferedReader, BufferedWriter)]
```

# Working with multiple resources

ScalaJobs

```
def  fileToRead(path: String): Resource[IO, BufferedReader] = ...
def fileToWrite(path: String): Resource[IO, BufferedWriter] = ...

import cats.implicits._

(fileToRead("file1.txt"), fileToWrite("file1-copy.txt"))
  .tupled
// res: Resource[IO, (BufferedReader, BufferedWriter)]
```

# Working with multiple resources

ScalaJobs

```
def  fileToRead(path: String): Resource[IO, BufferedReader] = ...
def fileToWrite(path: String): Resource[IO, BufferedWriter] = ...

import cats.implicits._

(fileToRead("file1.txt"), fileToWrite("file1-copy.txt"))
  .tupled
  .use { case (reader, writer) =>
    ???
  }
```

# Working with multiple resources

ScalaJobs

```
def  fileToRead(path: String): Resource[IO, BufferedReader] = ...
def fileToWrite(path: String): Resource[IO, BufferedWriter] = ...

import cats.implicits._

(fileToRead("file1.txt"), fileToWrite("file1-copy.txt"))
  .tupled
  .use { case (reader, writer) =>
    IO {
      Iterator
        .continually(reader.readLine())
        .takeWhile(_ != null)
        .foreach{ line =>
          writer.write(line)
          writer.newLine()
        }
    }
  }
```

# Library support

ScalaJobs

## Http4s

```
EmberServerBuilder
  .default[IO]
  .withHost(ipv4"0.0.0.0")
  .withPort(port"8080")
  .build
// res: Resource[IO, Server]
```

## Doobie

```
HikariTransactor.newHikariTransactor[IO](
  driverClassName = "org.postgresql.Driver",
  url             = "postgresql://localhost:5432/mydb",
  user            = "admin",
  pass            = "admin",
  connectEC       = ExecutionContext.global
)
// res: Resource[IO, HikariTransactor[IO]]
```

# Library support

ScalaJobs

## Http4s

```
EmberServerBuilder
  .default[IO]
  .withHost(ipv4"0.0.0.0")
  .withPort(port"8080")
  .build
// res: Resource[IO, Server]
```

## Doobie

```
HikariTransactor.newHikariTransactor[IO](
  driverClassName = "org.postgresql.Driver",
  url             = "postgresql://localhost:5432/mydb",
  user            = "admin",
  pass            = "admin",
  connectEC       = ExecutionContext.global
)
// res: Resource[IO, HikariTransactor[IO]]
```

# Library support

ScalaJobs

## Http4s

```
EmberServerBuilder
  .default[IO]
  .withHost(ipv4"0.0.0.0")
  .withPort(port"8080")
  .build
// res: Resource[IO, Server]
```

## Doobie

```
HikariTransactor.newHikariTransactor[IO](
  driverClassName = "org.postgresql.Driver",
  url             = "postgresql://localhost:5432/mydb",
  user            = "admin",
  pass            = "admin",
  connectEC       = ExecutionContext.global
)
// res: Resource[IO, HikariTransactor[IO]]
```

# Example 4: Queue

ScalaJobs

1. Pull an event from a queue
2. Process the event
3. On error, requeue the event

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  ...
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  ...
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  ...
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  ...
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    ...
  )
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    ...
  )
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    ...
  )
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    ...
  )
```

```
enum ExitCase {
  case Succeeded
  case Errored(e: Throwable)
  case Canceled
}
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    ...
  )
```

```
enum ExitCase {
  case Succeeded
  case Errored(e: Throwable)
  case Canceled
}
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    ...
  )
```

```
enum ExitCase {
  case Succeeded
  case Errored(e: Throwable)
  case Canceled
}
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    ...
  )
```

```
enum ExitCase {
  case Succeeded
  case Errored(e: Throwable)
  case Canceled
}
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue
import cats.effect.kernel.Resource.ExitCase._

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    exitCase match {
      case Succeeded             => ???
      case Errored(_) | Canceled => ???
    }
  )
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue
import cats.effect.kernel.Resource.ExitCase._

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    exitCase match {
      case Succeeded              => IO.unit
      case Errored(_) | Canceled => ???
    }
  )
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue
import cats.effect.kernel.Resource.ExitCase._

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    exitCase match {
      case Succeeded              => IO.unit
      case Errored(_) | Canceled => IO(queue.enqueue(value))
    }
  )
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue
import cats.effect.kernel.Resource.ExitCase._

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    exitCase match {
      case Succeeded             => IO.unit
      case Errored(_) | Canceled => IO(queue.enqueue(value))
    }
  )
  .evalTap(value => IO(println(s"Processing $value")))
  .onFinalize(IO(println("Queue after finalizer: " + queue)))
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue
import cats.effect.kernel.Resource.ExitCase._

def requeueOnError[A](queue: Queue[A]): Resource[IO, A] =
  Resource.makeCase(
    IO(queue.dequeue())
  )((value, exitCase) =>
    exitCase match {
      case Succeeded             => IO.unit
      case Errored(_) | Canceled => IO(queue.enqueue(value))
    }
  )
  .evalTap(value => IO(println(s"Processing $value")))
  .onFinalize(IO(println("Queue after finalizer: " + queue)))
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1.unsafeRunSync()
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1.unsafeRunSync()
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1.unsafeRunSync()
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1.unsafeRunSync()
```

# Queue

ScalaJobs

```
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1.unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1.unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1
  .replicateA(3)
  .unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
// Processing 2
// Queue after finalizer: Queue(3, 4, 2)
// Exception in thread "main" java.lang.Exception: Boom
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1
  .replicateA(3)
  .unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
// Processing 2
// Queue after finalizer: Queue(3, 4, 2)
// Exception in thread "main" java.lang.Exception: Boom
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1
  .replicateA(3)
  .unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
// Processing 2
// Queue after finalizer: Queue(3, 4, 2)
// Exception in thread "main" java.lang.Exception: Boom
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1
  .replicateA(3)
  .unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
// Processing 2
// Queue after finalizer: Queue(3, 4, 2)
// Exception in thread "main" java.lang.Exception: Boom
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1
  .handleErrorWith(e => IO(println(s"An error occurred: $e")))
  .replicateA(3)
  .unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
// Processing 2
// Queue after finalizer: Queue(3, 4, 2)
// An error occurred: java.lang.Exception: Boom
// Processing 3
// Queue after finalizer: Queue(4, 2)
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1
  .handleErrorWith(e => IO(println(s"An error occurred: $e")))
  .replicateA(3)
  .unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
// Processing 2
// Queue after finalizer: Queue(3, 4, 2)
// An error occurred: java.lang.Exception: Boom
// Processing 3
// Queue after finalizer: Queue(4, 2)
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1
  .handleErrorWith(e => IO(println(s"An error occurred: $e")))
  .replicateA(3)
  .unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
// Processing 2
// Queue after finalizer: Queue(3, 4, 2)
// An error occurred: java.lang.Exception: Boom
// Processing 3
// Queue after finalizer: Queue(4, 2)
```

# Queue

ScalaJobs

```scala
import scala.collection.mutable.Queue

val queue: Queue[Int] = Queue(1,2,3,4)

def process(event: Int): IO[Unit] =
  if(event == 2) IO.raiseError(new Exception("Boom"))
  else IO.unit

val consume1: IO[Unit] = requeueOnError(queue).use(process)

consume1
  .handleErrorWith(e => IO(println(s"An error occurred: $e")))
  .replicateA(3)
  .unsafeRunSync()
// Processing 1
// Queue after finalizer: Queue(2, 3, 4)
// Processing 2
// Queue after finalizer: Queue(3, 4, 2)
// An error occurred: java.lang.Exception: Boom
// Processing 3
// Queue after finalizer: Queue(4, 2)
```

# Resource summary

ScalaJobs

1. General solution
2. Support Sync and Async execution
3. Lots of utility methods and constructors
4. 3rd party libraries can expose them
5. Main drawback: it works with an IO-like effect

# Decision tree

ScalaJobs

If you use Vanilla Scala and/or Akka

Shutdown hooks and Using

# Decision tree

ScalaJobs

If you use Vanilla Scala and/or Akka

Shutdown hooks and Using

If you use Typelevel

Resource

# Decision tree

ScalaJobs

If you use Vanilla Scala and/or Akka

If you use Typelevel

If you use ZIO

Shutdown hooks and Using

Resource

Scope

ScalaJobs

# Thank You!

julien@scalajobs.com

ScalaJobs

# One last thing

# FS2 Stream

ScalaJobs

```
Files.forIO.readAll("users.csv")              // Stream of Byte
  .through(text.utf8.decode)                  // Stream of String
  .map(parseCsvLine)                          // Stream of Option[User]
  .unNone                                     // Stream of User
  .map(encodeToJson)                          // Stream of String
  .through(text.utf8.encode)                  // Stream of Byte
  .through(Files.forIO.writeAll("users.json")) // Stream of Nothing
  .compile.drain                              // IO     of Unit

case class User(name: String, email: String, approved: Boolean)

def parseCsvLine(line: String): Option[User] = ...

def encodeToJson(user: User): String = ...
```