Deep dive into Context Propagation

Jacob Wang

London Scala User Group, 2024 March

This talk

- Mechanisms for context propagation
- **otel4s** and **OpenTelemetry**

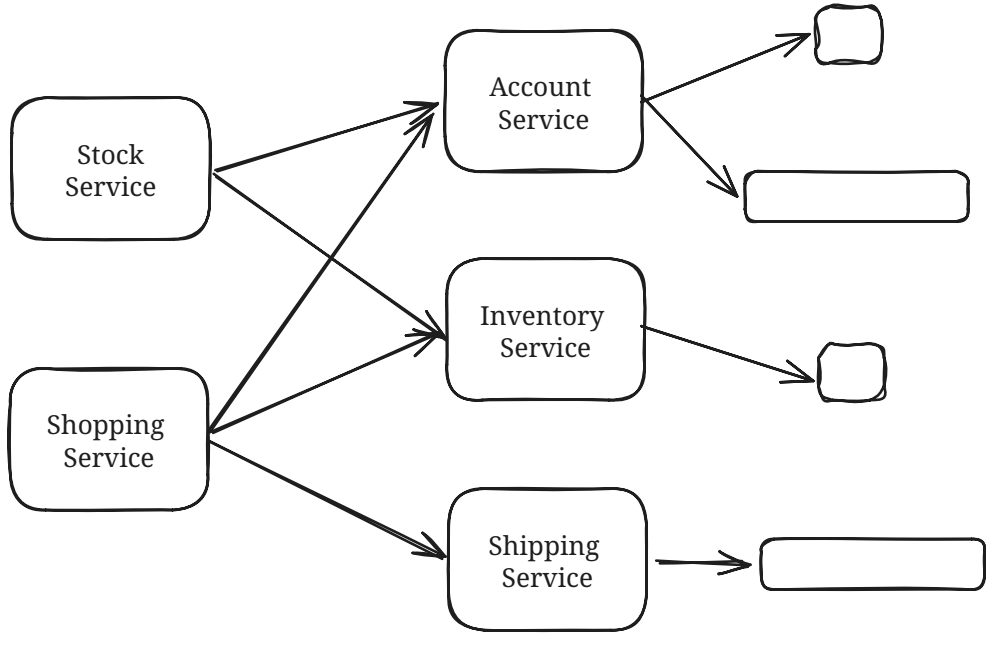
Hello

- Work at 

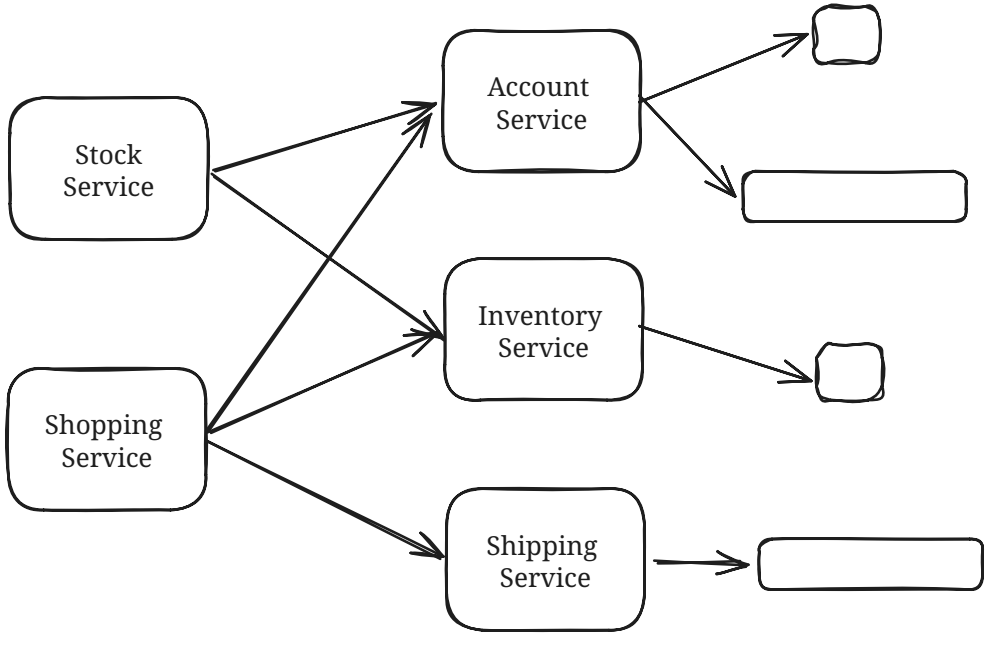
- Maintainer of Doobie and author of Difflicious + other libs
- https://mas.to/@jatcwang

Context Propagation - Large and Small

Context Propagation - Large and Small

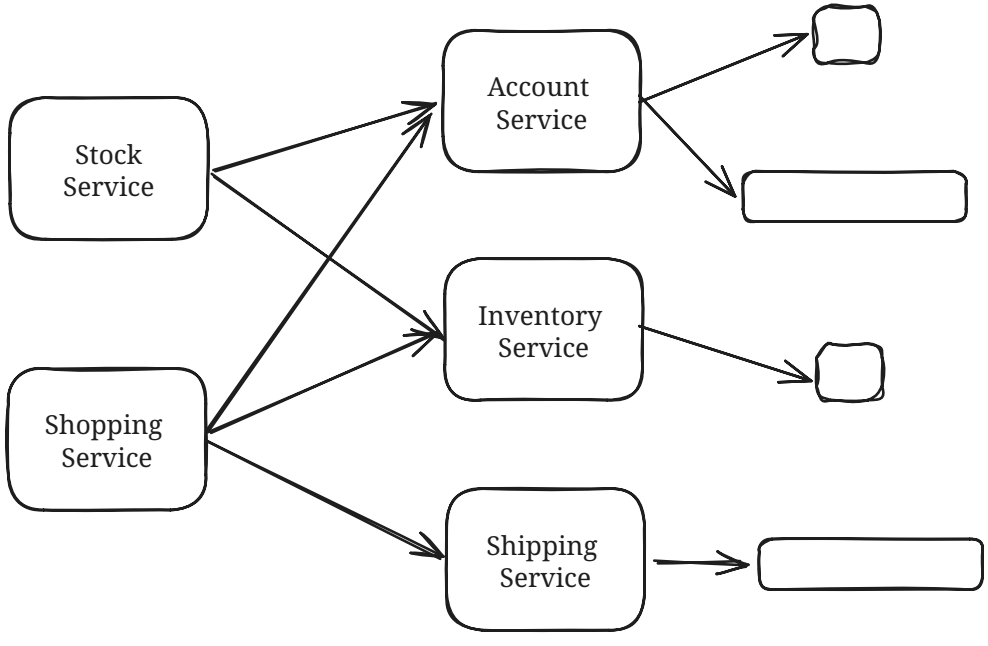


Context Propagation - Large and Small



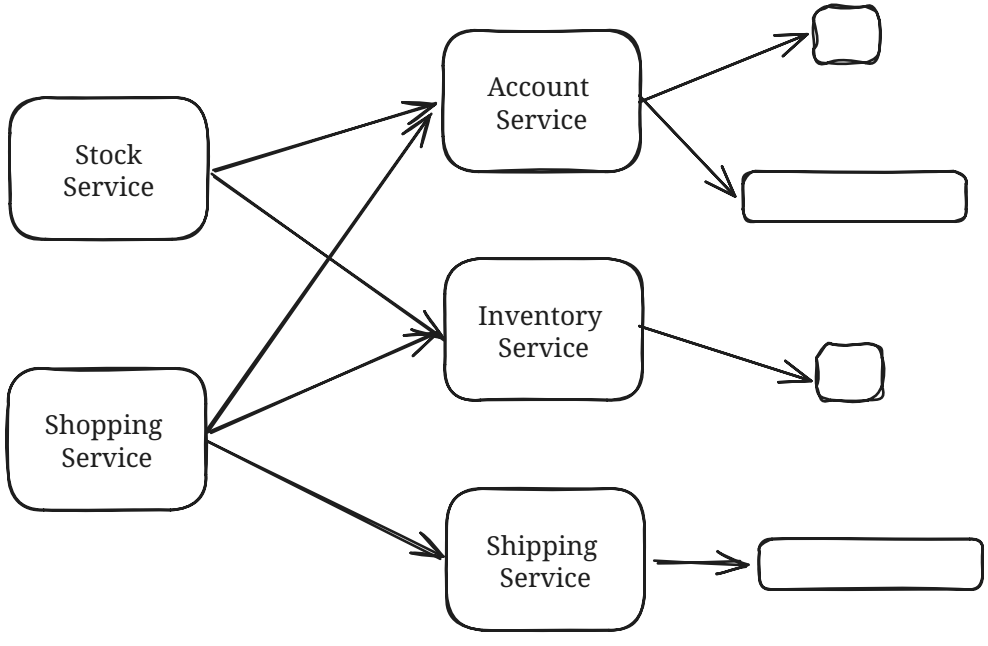
- Context: Not used for fulfilling the request, but instead for monitoring and debugging

Context Propagation - Large and Small



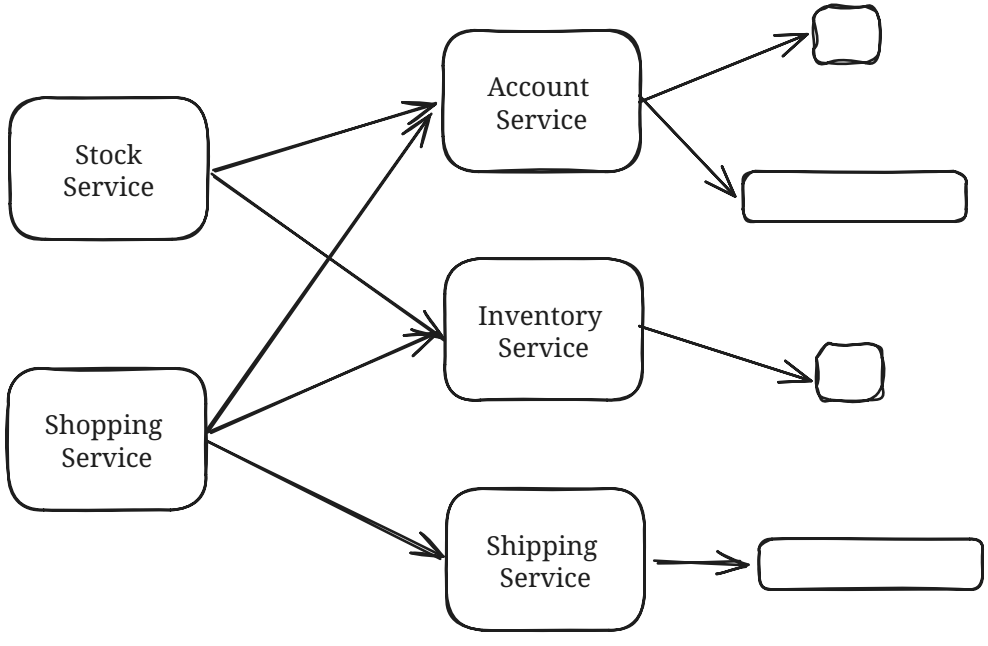
- Context: Not used for fulfilling the request, but instead for monitoring and debugging
  - e.g. Trace ID, Tenant ID, User ID

Context Propagation - Large and Small



- Context: Not used for fulfilling the request, but instead for monitoring and debugging
  - e.g. Trace ID, Tenant ID, User ID
- This talk: Passing context *within* an application

Context Propagation - Large and Small



- Context: Not used for fulfilling the request, but instead for monitoring and debugging
  - e.g. Trace ID, Tenant ID, User ID
- This talk: Passing context *within* an application
- Invisible / Non-local mechanisms only

java.lang.ThreadLocal

java.lang.ThreadLocal

- Store state **local** to the code executing on that thread only

java.lang.ThreadLocal

- Store state **local** to the code executing on that thread only
- ThreadLocal instances are “key” to set/remove their corresponding value stored in the thread

java.lang.ThreadLocal

- Store state **local** to the code executing on that thread only
- ThreadLocal instances are “key” to set/remove their corresponding value stored in the thread
- Conceptually, each Thread has a `Map[ThreadLocal[A], A]`

java.lang.ThreadLocal

- Store state **local** to the code executing on that thread only
- ThreadLocal instances are “key” to set/remove their corresponding value stored in the thread
- Conceptually, each Thread has a `Map[ThreadLocal[A], A]`

```
1  val TL_FOO: ThreadLocal[Int] = ThreadLocal.withInitial(() => 0)
2  val TL_BAR: ThreadLocal[Int] = ThreadLocal.withInitial(() => 0)
3
4  TL_FOO.set(5)
5
6  TL_FOO.get() // == 5
7  TL_BAR.get() // == 0, because TL_BAR is a different "key" from TL_FOO
8
9  TL_FOO.remove()
10
11 TL_FOO.get() // == 0
```

```
1  val t1 = new Thread(() => {
2    TL_FOO.get()   // == 0
3    TL_FOO.set(1)
4    TL_FOO.get()   // == 1
5  }, "t1").start()
6
7  val t2 = new Thread(() => {
8    TL_FOO.get()   // == 0
9    TL_FOO.set(3)
10   TL_FOO.get()   // == 3
11 }, "t2").start()
```

java.lang.ThreadLocal

- Allow us to pass context without explicit parameters

java.lang.ThreadLocal

- Allow us to pass context without explicit parameters
- Java logging **Mapped Diagnostic Context** (MDC) and **OpenTelemetry** pass their context using ThreadLocal

java.lang.ThreadLocal

- Allow us to pass context without explicit parameters
- Java logging **Mapped Diagnostic Context** (MDC) and **OpenTelemetry** pass their context using ThreadLocal
- Can use `InheritableThreadLocal` to pass on context to child threads

java.lang.ThreadLocal

- Allow us to pass context without explicit parameters
- Java logging **Mapped Diagnostic Context** (MDC) and **OpenTelemetry** pass their context using ThreadLocal
- Can use `InheritableThreadLocal` to pass on context to child threads
  - e.g. background tasks

The trouble with (many) Threads

The trouble with (many) Threads

- Old-school HTTP libraries create a new thread per request

The trouble with (many) Threads

- Old-school HTTP libraries create a new thread per request
- Each thread has a base cost of ~1MB

The trouble with (many) Threads

- Old-school HTTP libraries create a new thread per request
- Each thread has a base cost of ~1MB
- Threads are often blocked

The trouble with (many) Threads

- Old-school HTTP libraries create a new thread per request
- Each thread has a base cost of ~1MB
- Threads are often blocked
- CPU cores switching between threads (“context switching”) are expensive

Let’s reuse threads! 💡

Let’s reuse threads! 💡

- **threadpool**: pool of reusable threads and submit “tasks” (`Runnable`) to it

Let’s reuse threads! 💡

- **threadpool**: pool of reusable threads and submit “tasks” (`Runnable`) to it
- `Runnable` (equivalent to `() => Unit`) have much smaller overhead

Let’s reuse threads! 💡

- **threadpool**: pool of reusable threads and submit “tasks” (`Runnable`) to it
- `Runnable` (equivalent to `() => Unit`) have much smaller overhead
- “Async”

Let's reuse threads! 💡

- **threadpool**: pool of reusable threads and submit "tasks" (`Runnable`) to it
- `Runnable` (equivalent to `() => Unit`) have much smaller overhead
- "Async"
- Java/Scala Futures and Effect runtimes (Cats-effect, ZIO) are all built on top of threadpools

Let's reuse threads! 💡

- **threadpool**: pool of reusable threads and submit "tasks" (`Runnable`) to it
- `Runnable` (equivalent to `() => Unit`) have much smaller overhead
- "Async"
- Java/Scala Futures and Effect runtimes (Cats-effect, ZIO) are all built on top of threadpools

```
1  trait Runnable {
2    def run(): Unit
3  }
4
5  trait Executor { // The threadpool interface
6    def execute(task: Runnable): Unit
7  }
```

```scala
val threadpool = Executors.newFixedThreadPool(4)
val httpClient = ...
val finishCallback: Result => Unit = ...

threadpool.execute(() => {
  println(s"step 1")
  val httpRequest = ...

  httpClient.send(httpRequest, onResponse = response => {
    // Callback will submit next task to threadpool
    threadpool.execute(() => {
      val result = doWork(response)
      finishCallback(result)
    })

  })
})
```


I think I forgot something.
If you forgot, then it wasn't important.
Yeah, you're right.
ThreadLocal
imgflip.com

Losing Context

```
1    val threadpool = Executors.newFixedThreadPool(4) // 4 threads in pool
2    val CONTEXT = ThreadLocal.withInitial(() => "no user")
3
4    threadpool.execute(() => {
5      CONTEXT.set("user1")
6      println(s"${Thread.currentThread().getName}: step 1 for $
XT.get()}")
7
8      threadpool.execute(() => {
9        println(s"${Thread.currentThread().getName}: step 2 for $
XT.get()}")
10
11       threadpool.execute(() => {
12         println(s"${Thread.currentThread().getName}: complete for $
XT.get()}")
13       })
14
15     })
16   })
```

Losing Context

```
1    val threadpool = Executors.newFixedThreadPool(4) // 4 threads in pool
2    val CONTEXT = ThreadLocal.withInitial(() => "no user")
3
4    threadpool.execute(() => {
5      CONTEXT.set("user1")
6      println(s"${Thread.currentThread().getName}: step 1 for $
XT.get()}")
7
8      threadpool.execute(() => {
9        println(s"${Thread.currentThread().getName}: step 2 for $
XT.get()}")
10
11       threadpool.execute(() => {
12         println(s"${Thread.currentThread().getName}: complete for $
XT.get()}")
13       })
14
15     })
16   })
```

```
pool-1-thread-1: step 1 for user1
pool-1-thread-2: step 2 for no_user
pool-1-thread-3: complete for no_user
```

Losing Context

```
1    val threadpool = Executors.newFixedThreadPool(4) // 4 threads in pool
2    val CONTEXT = ThreadLocal.withInitial(() => "no user")
3
4    threadpool.execute(() => {
5      CONTEXT.set("user1")
6      println(s"${Thread.currentThread().getName}: step 1 for $
XT.get()}")
7
8      threadpool.execute(() => {
9        println(s"${Thread.currentThread().getName}: step 2 for $
XT.get()}")
10
11       threadpool.execute(() => {
12         println(s"${Thread.currentThread().getName}: complete for $
XT.get()}")
13       })
14
15     })
16   })
```

```
pool-1-thread-1: step 1 for user1
pool-1-thread-2: step 2 for no_user
pool-1-thread-3: complete for no_user
```

- Lost the context because **another thread** executed step 2

Losing Context

```
1    val threadpool = Executors.newFixedThreadPool(4) // 4 threads in pool
2    val CONTEXT = ThreadLocal.withInitial(() => "no user")
3
4    threadpool.execute(() => {
5      CONTEXT.set("user1")
6      println(s"${Thread.currentThread().getName}: step 1 for $
XT.get()}")
7
8      threadpool.execute(() => {
9        println(s"${Thread.currentThread().getName}: step 2 for $
XT.get()}")
10
11       threadpool.execute(() => {
12         println(s"${Thread.currentThread().getName}: complete for $
XT.get()}")
13       })
14
15     })
16   })
```

```
pool-1-thread-1: step 1 for user1
pool-1-thread-2: step 2 for no_user
pool-1-thread-3: complete for no_user
```

- Lost the context because **another thread** executed step 2
- Worse, another unrelated task executing on thread 1 now has “user 1” as its context!

Making ThreadLocal work with threadpools

- 💡 Capture ThreadLocal value in the submitting thread and set it when running in the new thread

Making ThreadLocal work with threadpools

- 💡 Capture ThreadLocal value in the submitting thread and set it when running in the new thread

```
1  class CurrentContextExecutor(delegate: Executor) extends Executor:
2    override def execute(task: Runnable): Unit =
3      val toAttach = CONTEXT.get()                          // T1
4      val wrappedTask = wrap(task, toAttach)                // T1
5      delegate.execute(wrappedTask)                         // T1
6
7    def wrap(task: Runnable, toAttach: Context): Runnable =
8      () =>
9        try                                                 //   T2
10         CONTEXT.set(toAttach)                             //   T2
11         task.run()                                        //   T2
12       finally                                             //   T2
13         CONTEXT.remove()                                  //   T2
```

Making ThreadLocal work with threadpools

- 💡 Capture ThreadLocal value in the submitting thread and set it when running in the new thread

```
1  class CurrentContextExecutor(delegate: Executor) extends Executor:
2    override def execute(task: Runnable): Unit =
3      val toAttach = CONTEXT.get()                          // T1
4      val wrappedTask = wrap(task, toAttach)                // T1
5      delegate.execute(wrappedTask)                         // T1
6
7    def wrap(task: Runnable, toAttach: Context): Runnable =
8      () =>
9        try                                                 //   T2
10         CONTEXT.set(toAttach)                             //   T2
11         task.run()                                        //   T2
12       finally                                             //   T2
13         CONTEXT.remove()                                  //   T2
```

  - Resurface the context we want to pass on (Line 3)

Making ThreadLocal work with threadpools

- 💡 Capture ThreadLocal value in the submitting thread and set it when running in the new thread

```
1  class CurrentContextExecutor(delegate: Executor) extends Executor:
2    override def execute(task: Runnable): Unit =
3      val toAttach = CONTEXT.get()                          // T1
4      val wrappedTask = wrap(task, toAttach)                // T1
5      delegate.execute(wrappedTask)                         // T1
6
7    def wrap(task: Runnable, toAttach: Context): Runnable =
8      () =>
9        try                                                 //   T2
10         CONTEXT.set(toAttach)                             //   T2
11         task.run()                                        //   T2
12       finally                                             //   T2
13         CONTEXT.remove()                                  //   T2
```

  - Resurface the context we want to pass on (Line 3)
  - Set the context we want to pass on (Line 10)

Making ThreadLocal work with threadpools

- 💡 Capture ThreadLocal value in the submitting thread and set it when running in the new thread

```
1  class CurrentContextExecutor(delegate: Executor) extends Executor:
2    override def execute(task: Runnable): Unit =
3      val toAttach = CONTEXT.get()                          // T1
4      val wrappedTask = wrap(task, toAttach)                // T1
5      delegate.execute(wrappedTask)                         // T1
6
7    def wrap(task: Runnable, toAttach: Context): Runnable =
8      () =>
9        try                                                 //   T2
10         CONTEXT.set(toAttach)                             //   T2
11         task.run()                                        //   T2
12       finally                                             //   T2
13         CONTEXT.remove()                                  //   T2
```

  - Resurface the context we want to pass on (Line 3)
  - Set the context we want to pass on (Line 10)
  - Remove the ThreadLocal value (Line 13)

cats.effect.IOLocal

cats.effect.IOLocal

- With Cats-Effect, *can* use `ThreadLocal` + task wrapping trick we just learned

cats.effect.IOLocal

- With Cats-Effect, *can* use `ThreadLocal` + task wrapping trick we just learned
- But we have a nicer solution: `IOLocal`

cats.effect.IOLocal

- With Cats-Effect, *can* use `ThreadLocal` + task wrapping trick we just learned
- But we have a nicer solution: `IOLocal`
- `IOLocal` allow us to pass context through the executing fiber

cats.effect.IOLocal

- With Cats-Effect, *can* use `ThreadLocal` + task wrapping trick we just learned
- But we have a nicer solution: `IOLocal`
- `IOLocal` allow us to pass context through the executing fiber
- On fork, child fibers inherit a copy of the parent's context

cats.effect.IOLocal Example

```
1  for
2    // Create the IOLocal "key"
3    CONTEXT <- IOLocal(default = 0)
4
5    _ <- CONTEXT.get                    // == 0
6    _ <- CONTEXT.set(5)
7
8    // fork!
9    fiber1 <- (for {
10     _ <- CONTEXT.get                  // == 5
11     _ <- CONTEXT.set(6)
12     _ <- CONTEXT.get                  // == 6
13   } yield ()).start
14
15   _ <- CONTEXT.get                    // == 5
16   _ <- CONTEXT.set(10)
17   _ <- fiber1.joinWithNever
18   _ <- CONTEXT.get                    // == 10
19 yield ()
```

What we’ve seen so far

What we’ve seen so far

- **ThreadLocal**

What we've seen so far

- **ThreadLocal**
- **Thread pools** and how ThreadLocal can work with them

What we've seen so far

- **ThreadLocal**
- **Thread pools** and how ThreadLocal can work with them
- `IOLocal` for cats-effect

What we've seen so far

- **ThreadLocal**
- **Thread pools** and how ThreadLocal can work with them
- `IOLocal` for cats-effect
- Now, let's apply what we've learned

OpenTelemetry - Quick Intro

- **OpenTelemetry**: An open standard for Distributed Tracing, Metrics, etc

OpenTelemetry - Quick Intro

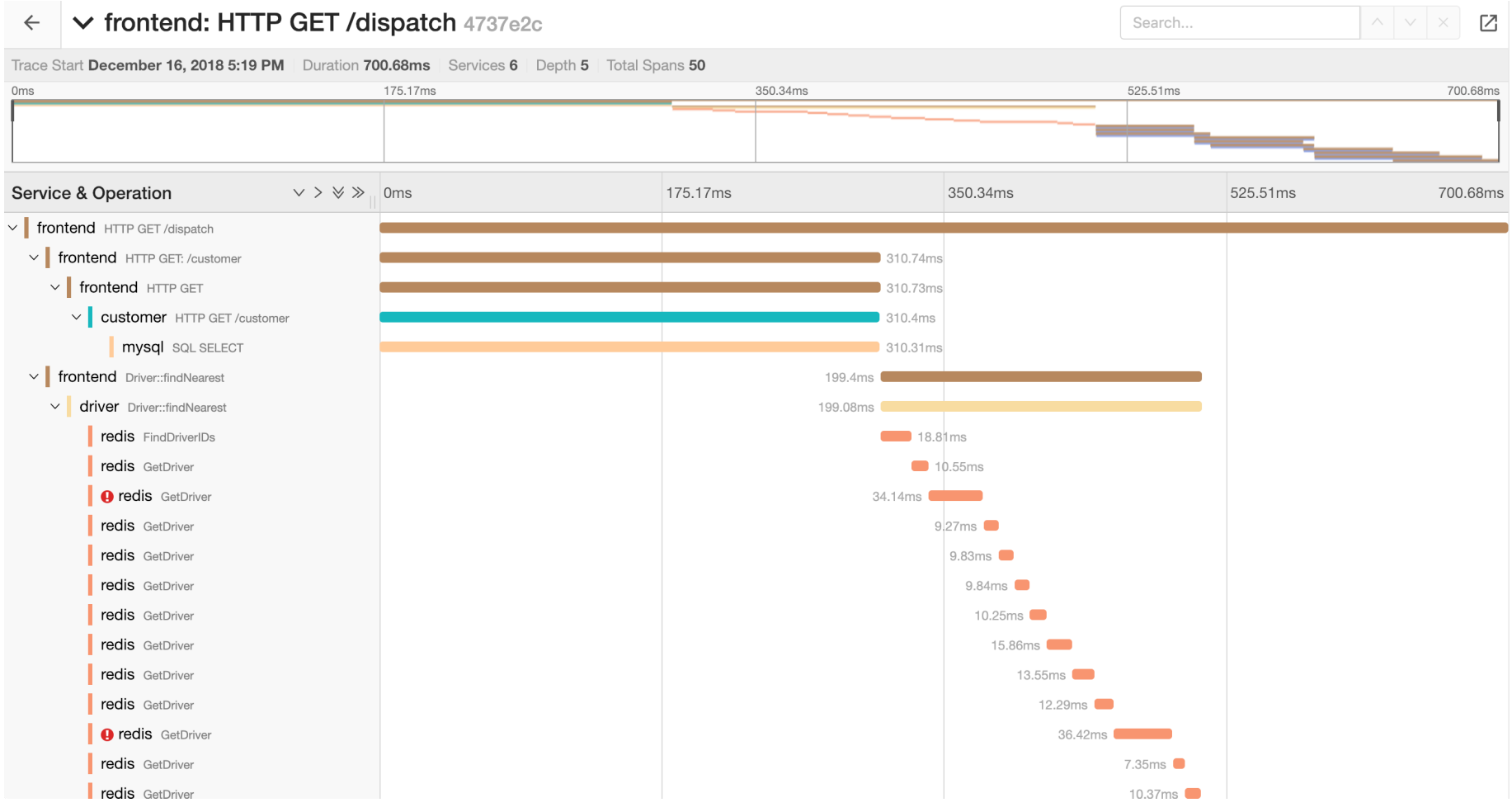
- **OpenTelemetry**: An open standard for Distributed Tracing, Metrics, etc
- **Span**: A recorded unit of work

OpenTelemetry - Quick Intro

- **OpenTelemetry**: An open standard for Distributed Tracing, Metrics, etc
- **Span**: A recorded unit of work
  - Attributes include `traceId`, `parentSpanId`, `isError` and `duration`

OpenTelemetry - Quick Intro

- **OpenTelemetry**: An open standard for Distributed Tracing, Metrics, etc
- **Span**: A recorded unit of work
  - Attributes include `traceId`, `parentSpanId`, `isError` and `duration`
- **Trace**: Links together a set of spans (same `traceId`), so we can track the execution trace from start to finish across services.


frontend: HTTP GET /dispatch 4737e2c
Search...
Trace Start December 16, 2018 5:19 PM
Duration 700.68ms
Services 6
Depth 5
Total Spans 50
0ms
175.17ms
350.34ms
525.51ms
700.68ms
Service & Operation
frontend HTTP GET /dispatch
frontend HTTP GET: /customer
frontend HTTP GET
customer HTTP GET /customer
mysql SQL SELECT
frontend Driver::findNearest
driver Driver::findNearest
redis FindDriverIDs
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
redis GetDriver
310.74ms
310.73ms
310.4ms
310.31ms
199.4ms
199.08ms
18.81ms
10.55ms
34.14ms
9.27ms
9.83ms
9.84ms
10.25ms
15.86ms
13.55ms
12.29ms
36.42ms
7.35ms
10.37ms

Otel4s - OpenTelemetry for Scala

Otel4s - OpenTelemetry for Scala

- Typelevel ecosystem

Otel4s - OpenTelemetry for Scala

- Typelevel ecosystem
- Follows OTel terminology and specification, but does not directly use OpenTelemetry-Java types

Otel4s - OpenTelemetry for Scala

- Typelevel ecosystem
- Follows OTel terminology and specification, but does not directly use OpenTelemetry-Java types
- Uses OpenTelemetry-Java as backend (e.g. reporting spans)

OpenTelemetry - Instrumentation API concepts

OpenTelemetry - Instrumentation API concepts

- `Context`: Immutable key-value pairs for storing all OTel-related objects

OpenTelemetry - Instrumentation API concepts

- `Context`: Immutable key-value pairs for storing all OTel-related objects
  - The `Context` object itself is propagated around (e.g. ThreadLocal, IOLocal)

OpenTelemetry - Instrumentation API concepts

- `Context`: Immutable key-value pairs for storing all OTel-related objects
  - The `Context` object itself is propagated around (e.g. ThreadLocal, IOLocal)
  - e.g. `Span` object is stored in `Context`

OpenTelemetry - Instrumentation API concepts

- `Context`: Immutable key-value pairs for storing all OTel-related objects
  - The `Context` object itself is propagated around (e.g. ThreadLocal, IOLocal)
  - e.g. `Span` object is stored in `Context`
  - ```
    val context = Map(
      SPAN_KEY -> Span(traceId, spanId, ...),
      BAGGAGE_KEY -> Baggage(..)
    )
    ```
    ```

OpenTelemetry - Instrumentation API concepts

- `Context`: Immutable key-value pairs for storing all OTel-related objects
  - The `Context` object itself is propagated around (e.g. ThreadLocal, IOLocal)
  - e.g. `Span` object is stored in `Context`
  - ```
    val context = Map(
      SPAN_KEY -> Span(traceId, spanId, ...),
      BAGGAGE_KEY -> Baggage(..)
    )
    ```
- `OpenTelemetry`/`Otel4s`: Central service for handling reporting spans and metrics
    ```

OpenTelemetry - Instrumentation API concepts

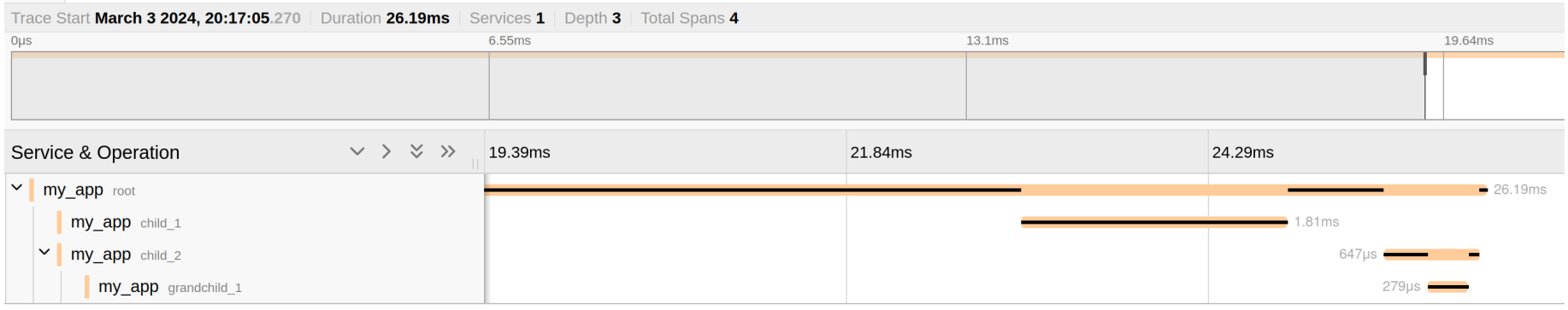
- `Context`: Immutable key-value pairs for storing all OTel-related objects
  - The `Context` object itself is propagated around (e.g. ThreadLocal, IOLocal)
  - e.g. `Span` object is stored in `Context`
  - ```
    val context = Map(
      SPAN_KEY -> Span(traceId, spanId, ...),
      BAGGAGE_KEY -> Baggage(..)
    )
    ```
- `OpenTelemetry`/`Otel4s`: Central service for handling reporting spans and metrics
- `Tracer`: Use to create spans. Created from `OpenTelemetry` instance
    ```

Otel4s - Minimal Example

```scala
import cats.effect._
import cats.implicits.*
import org.typelevel.otel4s.oteljava.context.Context
import org.typelevel.otel4s.oteljava.OtelJava
import org.typelevel.otel4s.trace.Tracer
```

```scala
for
  otel4s <- OtelJava.global[IO] // Initialize otel-java backend of Otel4s
  given Tracer[IO] <- otel4s.tracerProvider.get("my_app")

  _ <- Tracer[IO].span("root").surround(      // Start a span,
nding an IO
    for
      _ <- Tracer[IO].span("child_1").surround(
        IO.println("work work")
      )
      _ <- Tracer[IO].span("child_2").surround(
        Tracer[IO].span("grandchild_1").surround(
          IO.println("more work done")
        )
      )
    yield ()
  )
yield ()
```


Trace Start March 3 2024, 20:17:05.270
Duration 26.19ms
Services 1
Depth 3
Total Spans 4
0µs
6.55ms
13.1ms
19.64ms
Service & Operation
19.39ms
21.84ms
24.29ms
my_app root
my_app child_1
my_app child_2
my_app grandchild_1
26.19ms
1.81ms
647µs
279µs

Integrating with Java libraries

- In Otel4s, we use IOLocal to propagate context, but Java libs use opentelemetry-java (`ThreadLocal`)

Integrating with Java libraries

- In Otel4s, we use IOLocal to propagate context, but Java libs use opentelemetry-java (`ThreadLocal`)
- 💡 Extract `Context` from IOLocal and set it up as ThreadLocal before calling the java code

Integrating with Java libraries

- In Otel4s, we use IOLocal to propagate context, but Java libs use opentelemetry-java (`ThreadLocal`)
- 💡 Extract `Context` from IOLocal and set it up as ThreadLocal before calling the java code

```scala
import cats.mtl.Local
import io.opentelemetry.context.Context as JContext // Context type from Tel

def blockingWithContext[A](use: => A)(using local: Local[IO, Context]): =
  for
    context <- local.ask
    result <- IO.blocking {
      val jContext: JContext = context.underlying
      val scope = jContext.makeCurrent()       // Set the ThreadLocal
      try
        use
      finally
        scope.close()                          // Unset the ThreadLocal
    }
  yield result
```

Integrating with Java libraries - Example

Integrating with Java libraries - Example

- From cats-effect, use Java 11's `HttpClient` to call another service

Integrating with Java libraries - Example

- From cats-effect, use Java 11's `HttpClient` to call another service
- **Expectations**: Downstream service continues the trace

Integrating with Java libraries - Example

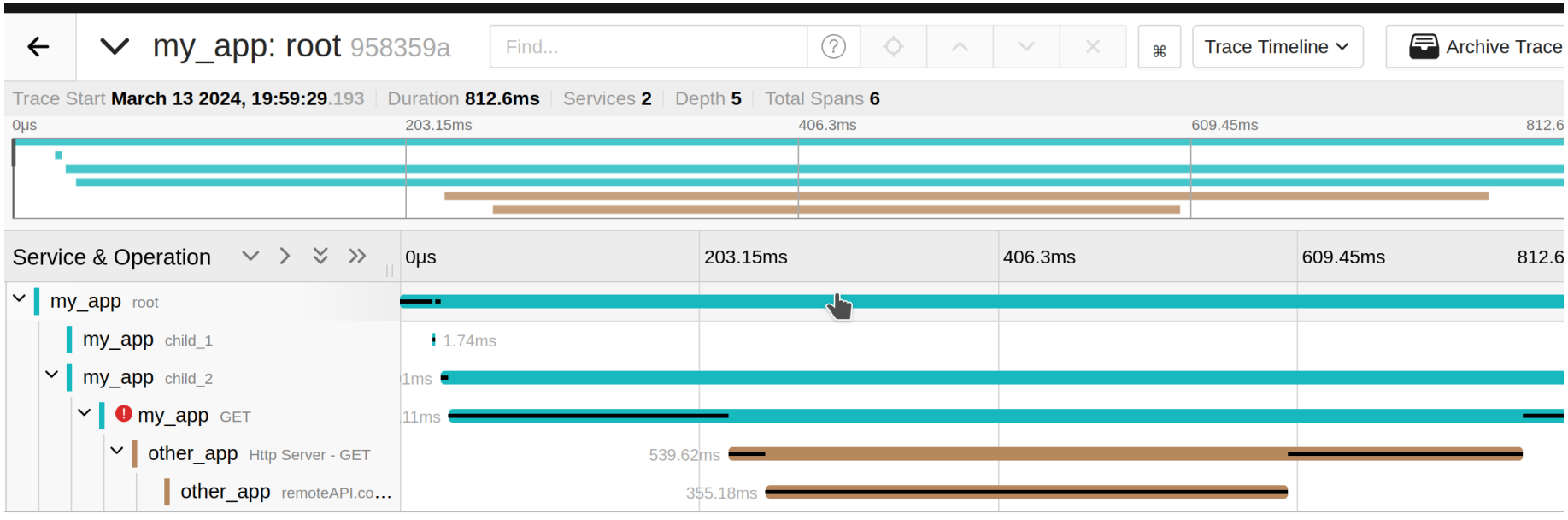
- From cats-effect, use Java 11's `HttpClient` to call another service
- **Expectations**: Downstream service continues the trace

```
      1  import
ntelemetry.instrumentation.httpclient.JavaHttpClientTelemetry
      2
      3  for
      4    otel4s <- OtelJava.global[IO]
      5    given Tracer[IO] <- otel4s.tracerProvider.get("my_app")
      6    given Local[IO, Context] = otel4s.localContext
      7
      8    httpClient = HttpClient.newBuilder().build()
      9    instrumentedHttpClient = JavaHttpClientTelemetry
     10            .builder(GlobalOpenTelemetry.get())
     11            .build()
     12            .newHttpClient(httpClient)
     13    myService = MyService(instrumentedHttpClient)
     14
     15    _ <- myService.doWorkAndMakeRequest
     16  yield ()
```

Integrating with Java libraries - Example

```
1  class MyService(javaHttpClient: HttpClient)(using Local[IO, Context],
[IO]):
2
3    def doWorkAndMakeRequest: IO[Unit] =
4      withSpan("root")(for
5        _ <- withSpan("child_1")(IO.println("work work"))
6
7        _ <- withSpan("child_2")(for
8          req <- IO(HttpRequest.newBuilder()
9            .uri(new URI("http://localhost:8080/example"))
10           .GET()
11           .build())
12
13         resp <- blockingWithContext {
14             javaHttpClient.send(req, BodyHandlers.ofString)
15           }
16         _ <- IO.println(resp.body())
17       yield ())
18
19     yield ())
20
21   def withSpan[F[_], A](name: String)(using tracer: Tracer[F])(io: F[A]):

22     tracer.span(name).surround(io)
```


my_app: root 958359a
Find...
Trace Timeline
Archive Trace
Trace Start March 13 2024, 19:59:29.193
Duration 812.6ms
Services 2
Depth 5
Total Spans 6
0µs
203.15ms
406.3ms
609.45ms
812.6
Service & Operation
my_app root
my_app child_1
1.74ms
my_app child_2
1ms
my_app GET
11ms
other_app Http Server - GET
539.62ms
other_app remoteAPI.co…
355.18ms

Final thoughts

- Otel4s supports metrics too!

Final thoughts

- Otel4s supports metrics too!
- use OTel-java's `Context.taskWrapping` to wrap threadpools

Final thoughts

- Otel4s supports metrics too!
- use OTel-java's `Context.taskWrapping` to wrap threadpools
  - Many async libraries allow you to pass in your own threadpool (`Executor`)

Final thoughts

- Otel4s supports metrics too!
- use OTel-java's `Context.taskWrapping` to wrap threadpools
  - Many async libraries allow you to pass in your own threadpool (`Executor`)
- Prefer Manual instrumentation? (Instead of using OTel Java agent)

Thanks to..

Thanks to..

- otel4s and OpenTelemetry

Thanks to..

- otel4s and OpenTelemetry
- https://github.com/keuhdall/otel4s-grafana-example

Thanks to..

- otel4s and OpenTelemetry
- https://github.com/keuhdall/otel4s-grafana-example
- Scala 3