

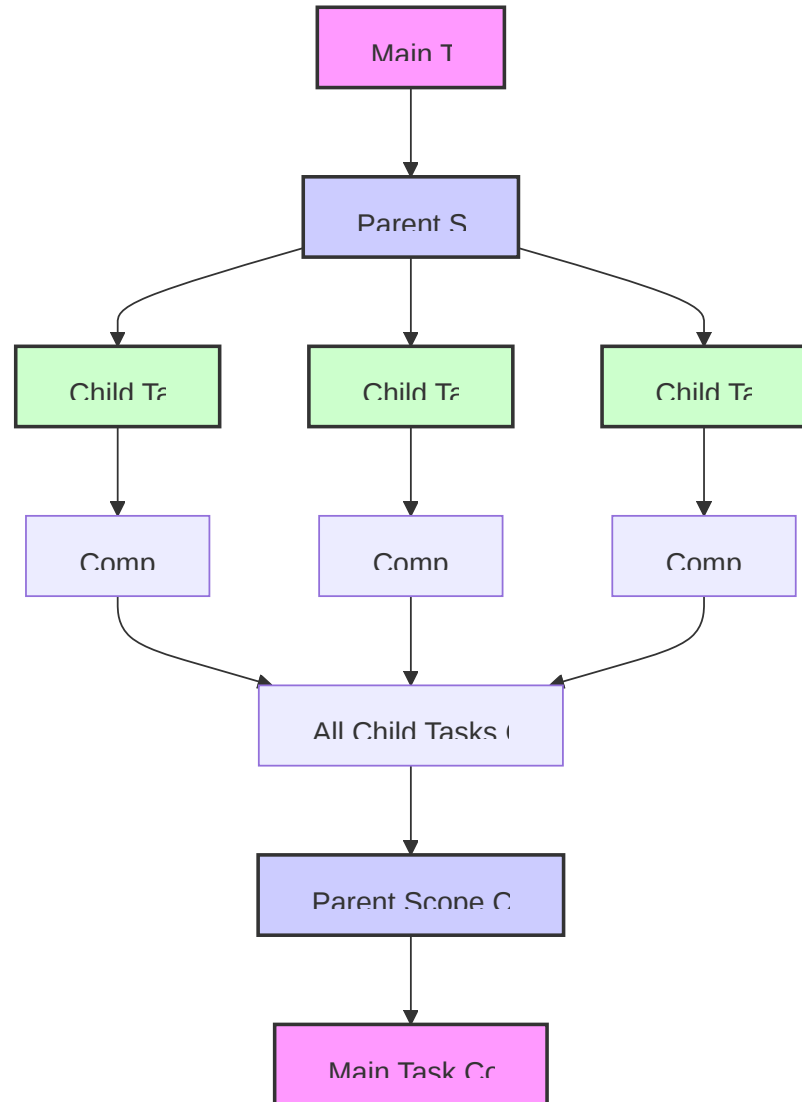
Structured Concurrency: Managing the Hierarchical Cancellation and Error Handling

James Ward

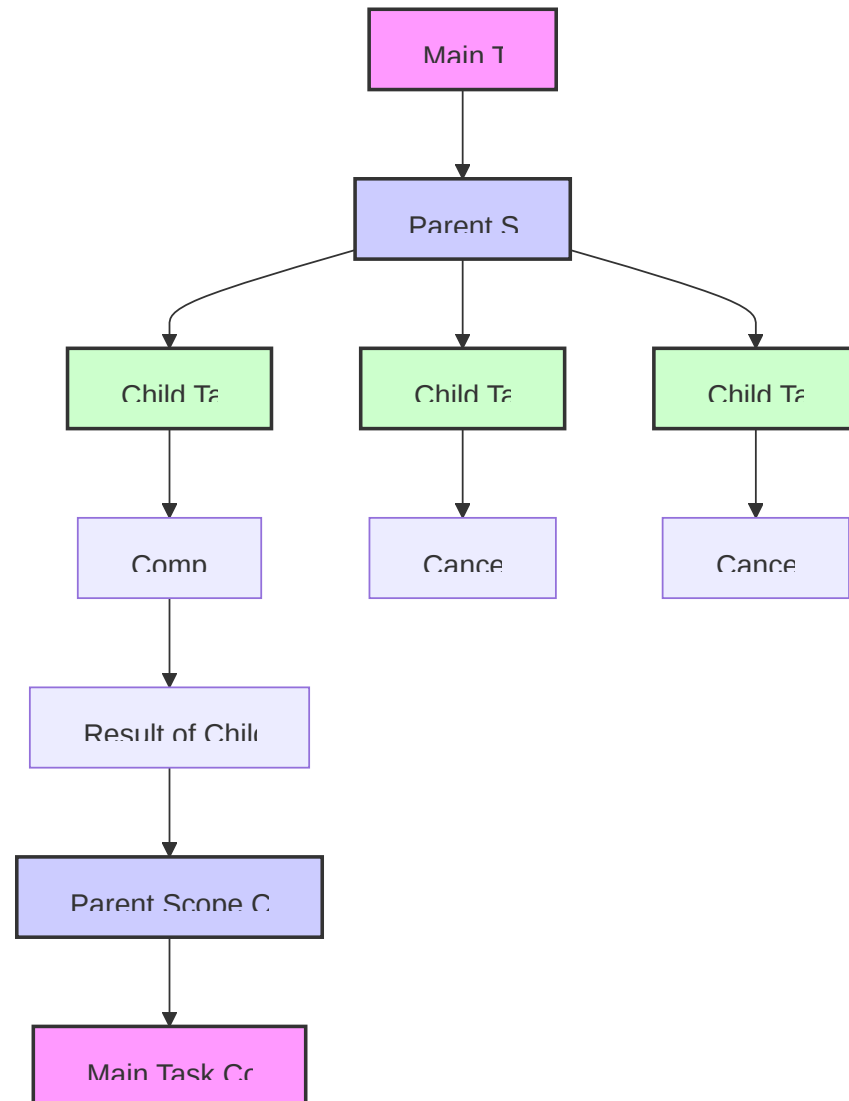
DX for Q Developer @ AWS

Follow [@_JamesWard](#)

What is Structured Concurrency?



Races



Easy Racer

github.com/jamesward/easyracer

Ten Structured Concurrency "obstacle courses"

Scala 3 + ZIO	Kotlin + Coroutines	OCaml + Lwt + Cohttp
Scala 3 + Ox	Kotlin + Splitties	OCaml + Eio + Cohttp
Scala 3 + Kyo	Kotlin + Arrow	Python (Various)
Scala + Cats Effects 3	Rust + Tokio	C#
Java + Loom	Go	Elm

Approaches to Structured Concurrency

- Scoped Driven
 - Java Loom (JEP 453)
- Direct Style (Imperative / Monad free!)
 - Jox & Scala Ox
 - Built on Loom, JDK21+ only
 - Rust (Future based syntax)
- Effect Oriented
 - Scala ZIO
 - Monadic Effect
 - Scala Kyo
 - Algebraic Effects / single monad

Scenario 1

Race 2 concurrent requests

Scenario 1 - Scala Ox

```
def scenario1(scenarioUrl: Int => Uri): String =  
  val url = scenarioUrl(1)  
  def req = scenarioRequest(url).send(backend).body  
  raceSuccess(req, req)
```

Scenario 1 - Java Jox

```
public String scenario1() throws ExecutionException, InterruptedException {  
    var req = HttpRequest.newBuilder(url.resolve("/1")).build();  
    return race(  
        () -> client.send(req, HttpResponse.BodyHandlers.ofString()),  
        () -> client.send(req, HttpResponse.BodyHandlers.ofString())  
    ).body();  
}
```

Scenario 1 - Java Loom

```
public String scenario1() throws ExecutionException, InterruptedException {  
    var req = HttpRequest.newBuilder(url.resolve("/1")).build();  
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {  
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));  
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));  
        scope.join();  
        return scope.result().body();  
    }  
}
```

Scenario 1 - Kotlin Coroutines

```
suspend fun scenario1(url: (Int) -> String): String = coroutineScope {  
    try {  
        select {  
            async { client.get(url(1)) }.onAwait { it }  
            async { client.get(url(1)) }.onAwait { it }  
        }.bodyAsText()  
    } finally {  
        coroutineContext.cancelChildren()  
    }  
}
```

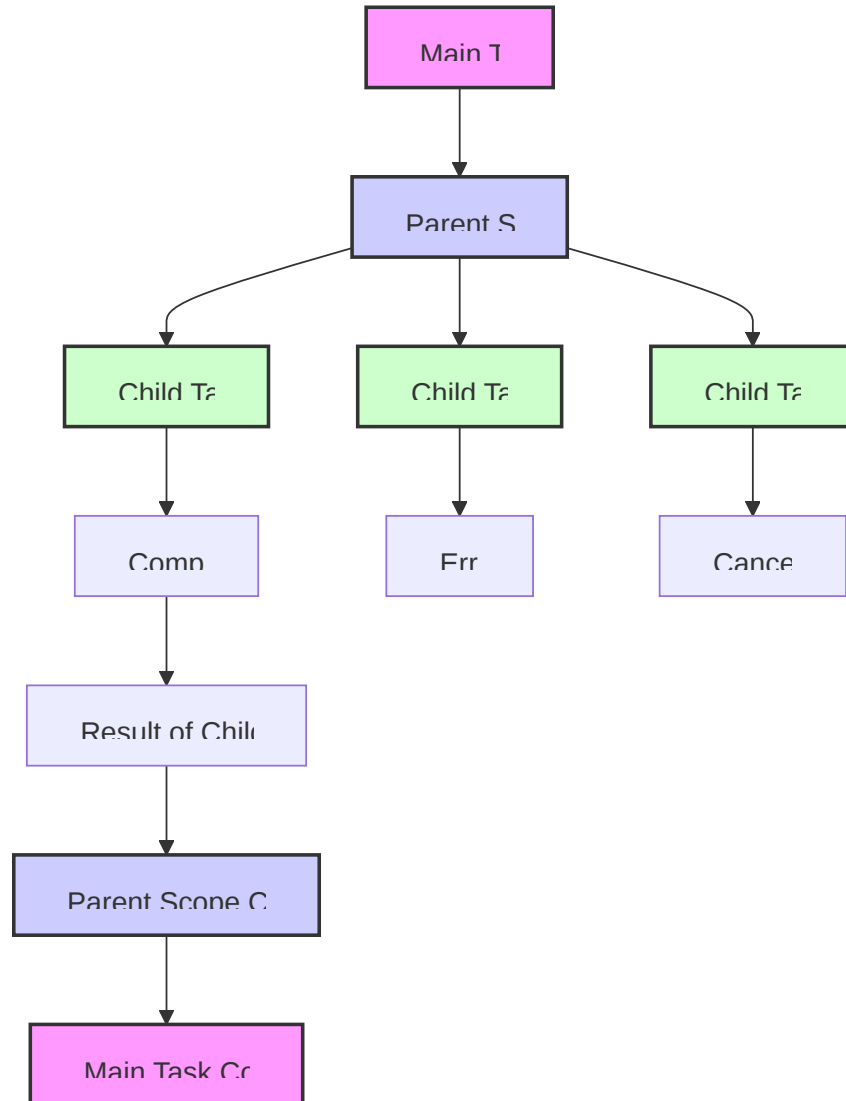
Scenario 1 - Scala ZIO

```
def scenario1(scenarioUrl: Int => String) =  
  defer:  
    val url = scenarioUrl(1)  
    val req = Client.batched(Request.get(url))  
    val winner = req.race(req).run  
    winner.body.asString.run
```

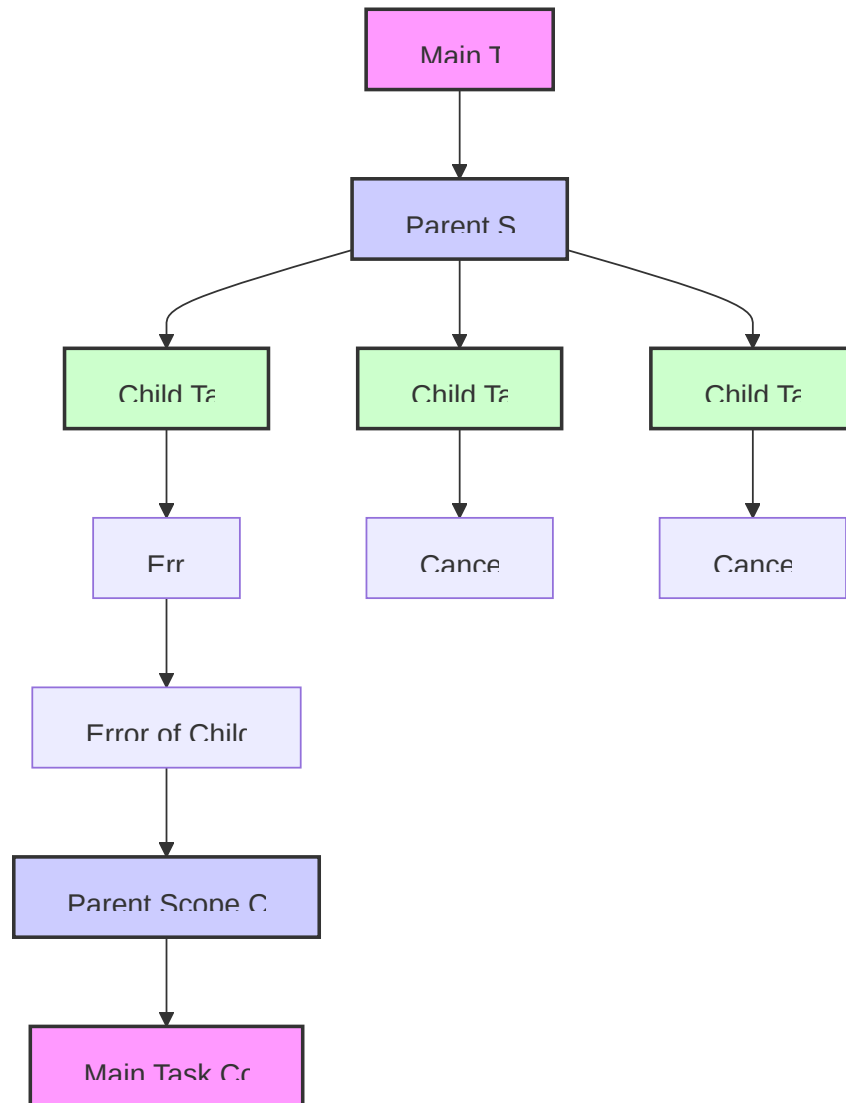
Scenario 2

Race 2 concurrent requests, where one produces a connection error

Race Shutdown on Success



Race Shutdown on Error



Scenario 2 - Java Loom

```
public String scenario2() throws ExecutionException, InterruptedException {  
    var req = HttpRequest.newBuilder(url.resolve("/2")).build();  
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {  
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));  
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));  
        scope.join();  
        return scope.result().body();  
    }  
}
```

Scenario 2 - Kotlin Splitties

```
suspend fun scenario2(url: (Int) -> String) =  
    raceOf({  
        try {  
            client.get(url(2))  
        } catch (e: Exception) {  
            awaitCancellation()  
        }  
    }, {  
        try {  
            client.get(url(2))  
        } catch (e: Exception) {  
            awaitCancellation()  
        }  
    })  
    .bodyAsText()
```

Scenario 3

Race 10,000 concurrent requests

Scenario 3 - Java Loom

```
public String scenario3() throws ExecutionException, InterruptedException {  
    var req = HttpRequest.newBuilder(url.resolve("/3")).build();  
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {  
        IntStream.rangeClosed(1, 10_000)  
            .forEach(i ->  
                scope.fork(() ->  
                    client.send(req, HttpResponse.BodyHandlers.ofString())  
                )  
            );  
  
        scope.join();  
        return scope.result().body();  
    }  
}
```

Scenario 3 - Java Jox

```
public String scenario3() throws ExecutionException, InterruptedException {  
    var req = HttpRequest.newBuilder(url.resolve("/3")).build();  
    List<Callable<HttpResponse<String>>> reqs =  
        Collections.nCopies(10_000, () ->  
            client.send(req, HttpResponse.BodyHandlers.ofString()))  
        );  
    return race(reqs).body();  
}
```

Scenario 4

Race 2 concurrent requests but 1 of them should have a 1 second timeout

Scenario 4 - Java Loom

```
public String scenario4() throws ExecutionException, InterruptedException {  
    var req = HttpRequest.newBuilder(url.resolve("/4")).build();  
    try (var outer = new StructuredTaskScope.ShutdownOnSuccess<String>()) {  
        outer.fork(() -> {  
            try (var inner = new StructuredTaskScope.ShutdownOnSuccess<String>()) {  
                inner.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()).body());  
                inner.joinUntil(Instant.now().plusSeconds(1));  
                return inner.result();  
            }  
        });  
  
        outer.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()).body());  
  
        outer.join();  
  
        return outer.result();  
    }  
}
```

Scenario 4 - Java Jox

```
public String scenario4() throws ExecutionException, InterruptedException {  
    var req = HttpRequest.newBuilder(url.resolve("/4")).build();  
    return race(  
        () -> client.send(req, HttpResponse.BodyHandlers.ofString()),  
        () -> timeout(1000, () -> client.send(req, HttpResponse.BodyHandlers.ofString()))  
    ).body();  
}
```

Scenario 7

Hedging: Start a request, wait at least 3 seconds then start a second request (hedging), and race the two

Scenario 7 - Java Loom

```
public String scenario7() throws ExecutionException, InterruptedException {  
    var req = HttpRequest.newBuilder(url.resolve("/7")).build();  
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {  
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));  
        scope.fork(() -> {  
            Thread.sleep(3000);  
            return client.send(req, HttpResponse.BodyHandlers.ofString());  
        });  
        scope.join();  
        return scope.result().body();  
    }  
}
```

Scenario 8

Race 2 concurrent requests that "use" a resource which is obtained and released through other requests. The "use" request can return a non-20x request, in which case it is not a winner.

Scenario 8 - Scala Ox

```
def scenario8(scenarioUrl: Int => Uri): String =  
  def req(url: Uri) = basicRequest.get(url).response(asString.getRight).send(backend).body  
  
  def open = req(uri"${scenarioUrl(8)}?open")  
  def use(id: String) = req(uri"${scenarioUrl(8)}?use=$id")  
  def close(id: String) = req(uri"${scenarioUrl(8)}?close=$id")  
  
  def reqRes = supervised:  
    val id = useInScope(open)(close)  
    use(id)  
  
  raceSuccess(reqRes, reqRes)
```

Scenario 8 - Java Loom

```
class Req implements AutoCloseable {
    final HttpRequest openReq =
        HttpRequest.newBuilder(url.resolve("/8?open")).build();
    final Function<String, HttpRequest> useReq = (id) ->
        HttpRequest.newBuilder(url.resolve("/8?use=" + id)).build();
    final Function<String, HttpRequest> closeReq = (id) ->
        HttpRequest.newBuilder(url.resolve("/8?close=" + id)).build();

    final String id;

    public Req() throws IOException, InterruptedException {
        id = client.send(openReq, HttpResponse.BodyHandlers.ofString()).body();
    }

    HttpResponse<String> make() throws Exception {
        var resp = client.send(useReq.apply(id), HttpResponse.BodyHandlers.ofString());
        if (resp.statusCode() == 200) {
            return resp;
        } else {
            throw new Exception("invalid response");
        }
    }

    @Override
    public void close() throws IOException, InterruptedException {
        client.send(closeReq.apply(id), HttpResponse.BodyHandlers.ofString()).body();
    }
}
```

Scenario 8 - Java Loom

```
try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
    scope.fork(() -> {
        try (var req = new Req()) {
            return req.make();
        }
    });
    scope.fork(() -> {
        try (var req = new Req()) {
            return req.make();
        }
    });

    scope.join();

    return scope.result().body();
}
```

Scenario 9

**Make 10 concurrent requests where 5 return a 200 response with a letter.
Assemble the letters in the order they were received.**

Scenario 9 - Java Loom

```
public String scenario9() throws InterruptedException {
    record TimedResponse(Instant instant, HttpResponse<String> response) {
    }

    final HttpRequest req = HttpRequest.newBuilder(url.resolve("/9")).build();
    try (var scope = new StructuredTaskScope<TimedResponse>()) {
        var futures = IntStream.rangeClosed(1, 10)
            .mapToObj(i ->
                scope.fork(() -> {
                    var resp = client.send(req, HttpResponse.BodyHandlers.ofString());
                    return new TimedResponse(Instant.now(), resp);
                })
            ).toList();

        scope.join();

        return futures.stream()
            .map(StructuredTaskScope.Subtask::get)
            .filter(r -> r.response.statusCode() == 200)
            .sorted(Comparator.comparing(TimedResponse::instant)).collect(
                StringBuilder::new,
                (acc, timedResponse) -> acc.append(timedResponse.response.body()),
                StringBuilder::append
            ).toString();
    }
}
```

Scenario 9 - Kotlin Coroutines

```
suspend fun scenario9(url: (Int) -> String): String = coroutineScope {  
    val letters = List(10) {  
        async {  
            client.get(url(9)).let {  
                if (it.status.isSuccess())  
                    Instant.now() to it.bodyAsText()  
                else null  
            }  
        }  
    }.awaitAll()  
  
    letters.filterNotNull().sortedBy { it.first }.joinToString("") { it.second }  
}
```

So what?

- Concurrency is hard
- Higher level abstractions are helpful
- Effects: Most flexibility (Race anything, Resource Management, Standard Error Handling, Abstraction over Scheduler, OOTB High-Level Combinators)