Wicked Fast API Calls With ZIO Query

Introduction

ZIO Query is a library for writing optimized queries to data sources in a high level, compositional style

- Based on **"There is no Fork: an Abstraction for Efficient, Concurrent, and Concise Data Access"** by Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy
- Similar to **HAXL** library in Haskell by Facebook
- But also significant differences

The Problem

```
val getAllUserIds: ZIO[Any, Nothing, List[Int]]          = ???
def getUserNameById(id: Int): ZIO[Any, Nothing, String] = ???

for {
  userIds   <- getAllUserIds
  userNames <- ZIO.foreachPar(userIds)(getUserNameById)
} yield userNames
```

- Composable
- General
- But not performant

Not A Solution

```
lazy val getAllUserIds: ZIO[Any, Nothing, List[Int]]                = ???
def getUserNamesById(ids: List[Int]): ZIO[Any, Nothing, List[String]] = ???

for {
  userIds   <- getAllUserIds
  userNames <- getUserNamesById(userIds)
} yield userNames
```

- Performant
- General
- But not composable

A Solution

```
val getAllUserIds: ZQuery[Any, Nothing, List[Int]]          = ???
def getUserNameById(id: Int): ZQuery[Any, Nothing, String] = ???

for {
  userIds   <- getAllUserIds
  userNames <- ZQuery.foreachPar(userIds)(getUserNameById)
} yield userNames
```

- Composable
- General
- Performant

Batching

Approach

Separate the description and interpretation of a query

- Capture queries as data reflecting parallel and sequential structure
- Evaluate everything that can be performed in parallel as a batch
- Repeat sequentially

ZQuery

```
trait ZQuery[-R, +E, +A] {
  def step: ZIO[(R, QueryContext), Nothing, Result[R, E, A]]
}
```

A **`ZQuery`** is an effect that returns a **`Result`** which may be either:

1. Done with a value **`A`**
2. Failed with an **`E`**
3. Blocked on requests to data sources with a continuation

Result

```
sealed trait Result[-R, +E, +A]

final case class Blocked[-R, +E, +A](
  blockedRequests: BlockedRequests[R],
  continue: ZQuery[R, E, A]
) extends Result[R, E, A]
final case class Done[+A](value: A) extends Result[Any, Nothing, A]
final case class Fail[+E](cause: Cause[E]) extends Result[Any, E, Nothing]
```

Sequential Queries

```
final def flatMap[R1 <: R, E1 >: E, B](f: A => ZQuery[R1, E1, B]): ZQuery[R1, E1, B] =
  ZQuery {
    step.flatMap {
      case Result.Blocked(br, c) => ZIO.succeedNow(Result.blocked(br, c.flatMap(f)))
      case Result.Done(a)        => f(a).step
      case Result.Fail(e)        => ZIO.succeedNow(Result.fail(e))
    }
  }
```

If this query is blocked on a requests to data sources we are still blocked and simply add to the continuation

Parallel Queries

```
final def zipWithPar[R1 <: R, E1 >: E, B, C](that: ZQuery[R1, E1, B])(f: (A, B) => C): ZQuery[R1, E1, C] =
  ZQuery {
    self.step.zipWithPar(that.step) {
      case (Result.Blocked(br1, c1), Result.Blocked(br2, c2)) => Result.blocked(br1 ++ br2, c1.zipWithPar(c2)(f))
      case (Result.Blocked(br, c), Result.Done(b))             => Result.blocked(br, c.map(a => f(a, b)))
      case (Result.Done(a), Result.Blocked(br, c))             => Result.blocked(br, c.map(b => f(a, b)))
      case (Result.Done(a), Result.Done(b))                    => Result.done(f(a, b))
      case (Result.Fail(e1), Result.Fail(e2))                  => Result.fail(Cause.Both(e1, e2))
      case (Result.Fail(e), _)                                 => Result.fail(e)
      case (_, Result.Fail(e))                                 => Result.fail(e)
    }
  }
```

If both queries are blocked on requests to data sources we can combine them into one query that is blocked on both sets of requests and combines their continuations

Requests

```
trait Request[+E, +A]
```

A request to a data source that may fail with an **E** or succeed with an **A**

- Users extend to describe the requests their data sources support
- Purely a description of what is being requested
- No actual logic for how request should be executed

Data Sources

```
trait DataSource[-R, -A] {
  def run(requests: Chunk[A]): ZIO[R, Nothing, CompletedRequestMap]
}
```

- Able to execute requests of type **A** using an environment **R**
- Parameterized on a collection of requests so can batch
- Supports polymorphic requests in type safe way

Query Constructors

```
def fromRequest[R, E, A, B](
  request: A
)(dataSource: DataSource[R, A])(implicit ev: A <:< Request[E, B]): ZQuery[R, E, B]

def succeed[A](a: => A): ZQuery[Any, Nothing, A]

def fromEffect[R, E, A](zio: ZIO[R, E, A]): ZQuery[R, E, A]
```

- **`fromRequest`** - from a request and a data source
- **`succeed`** - from a pure value
- **`fromEffect`** - from an effect

Creating Data Sources

```scala
trait UserRequest[+A] extends Request[Nothing, A]

case object GetAllUserIds extends UserRequest[List[Int]]
final case class GetUserNameById(id: Int) extends UserRequest[String]

val userDataSource: DataSource[Any, UserRequest] =
  new DataSource[Any, UserRequest] {
    def run(requests: Chunk[UserRequest]): ZIO[Any, Nothing, CompletedRequestMap] =
      ???
  }
```

Creating Requests

```
val getAllUserIds: ZQuery[Any, Nothing, List[Int]] =
  ZQuery.fromRequest(GetAllUserIds)(userRequestDataSource)

def getUserNameById(id: Int): ZQuery[Any, Nothing, String] =
  ZQuery.fromRequest(GetUserNameById(id))(userRequestDataSource)
```

Running Queries

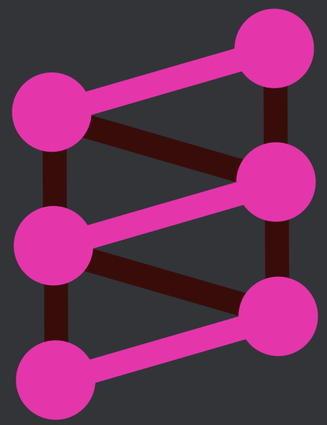
```
val query: ZQuery[Any, Nothing, List[String]] =
  for {
    userIds   <- getAllUserIds
    userNames <- ZQuery.foreachPar(userIds)(getUserNameById)
  } yield userNames

val zio: ZIO[Any, Nothing, List[String]] =
  query.run
```

Caliban

Next generation GraphQL library for Scala

- Automatic derivation of schemas from data types
- Query parsing and validation
- Effects handled by ZIO

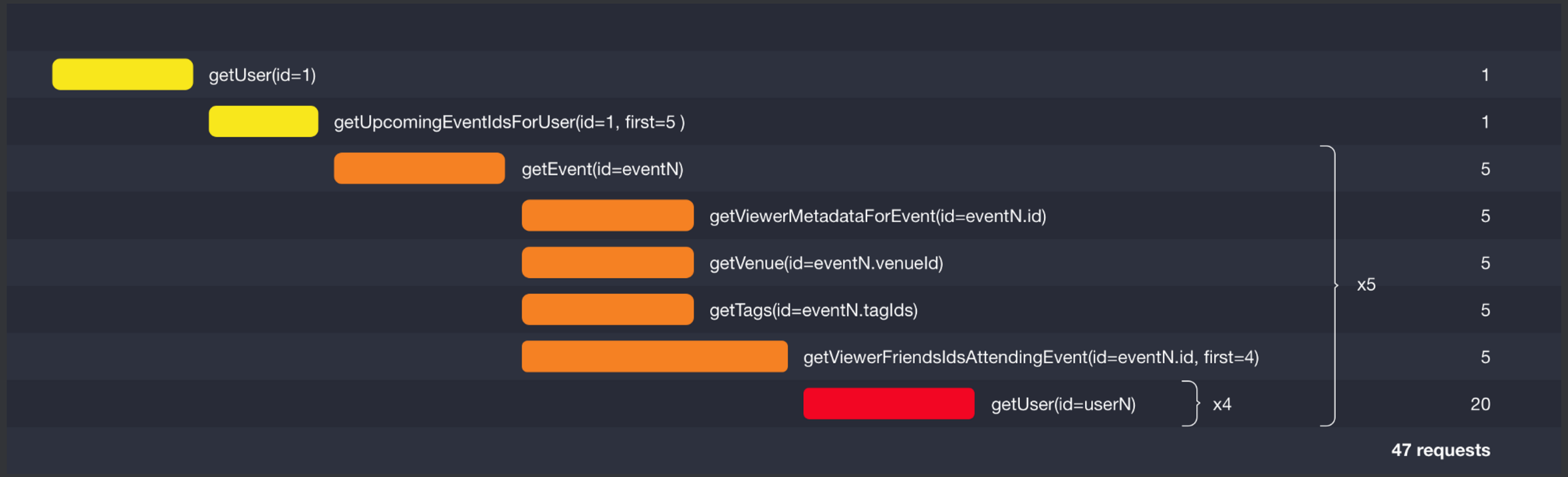


Using ZQuery With Caliban

```
case class Queries(
  users: ZQuery[Any, Nothing, List[User]],
  user: UserArgs => ZQuery[Any, Nothing, User])
```

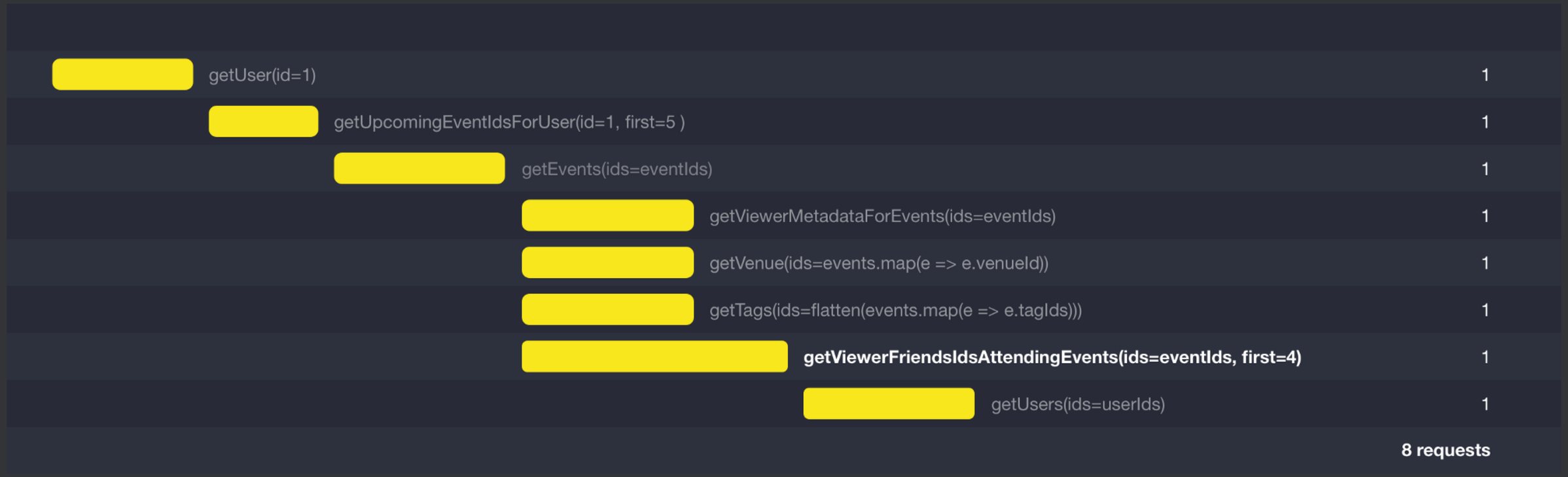
- Fields can return effects
- Effects will be run ever time a query requiring the field is executed
- To optimize queries, simply include fields of type **`ZQuery`** in your API definition

Unoptimized Query



"Optimizing Your GraphQL Request Waterfalls" by Andy Ingram

Optimized Query



"Optimizing Your GraphQL Request Waterfalls" by Andy Ingram

Pipelining

Models of Computation

So far we have considered two models of computation

1. Sequential and dependent computations
2. Parallel and independent computations

Sequential and Dependent

```
for {
  userIds   <- getAllUserIds
  userNames <- getUserNamesById(userIds)
} yield userNames
```

- expressed with **flatMap**
- don't know second query until first is complete
- minimal possibilities for optimization

Parallel and Independent

```
ZQuery.foreachPar(userIds)(getUserNameById)
```

- Expressed with **`zipWithPar`**
- Both queries known in advance and can be performed in parallel
- We can just batch them

Sequential and Independent

```
incr("some_key") *> ping *> decr("some_other_key")
```

- Expressed using **`zipWith`**
- Both queries known in advance but must be performed sequentially
- How do we pipeline them?

Approach

Capture more information in our description of a query

- Preserve parallel and sequential structure in blocked requests
- Determine requests that can be safely pipelined
- Pass requests that can be optimized to data sources

Blocked Requests

```
sealed trait BlockedRequests[-R]

final case class Single[-R, A](
  dataSource: DataSource[R, A],
  blockedRequest: BlockedRequest[A]
) extends BlockedRequests[R]
final case class Both[-R](left: BlockedRequests[R], right: BlockedRequests[R])
  extends BlockedRequests[R]
final case class Then[-R](left: BlockedRequests[R], right: BlockedRequests[R])
  extends BlockedRequests[R]
```

A free semiring that losslessly preserves parallel and sequential requests, similar to **`Cause`** in ZIO

Running Requests

We can safely pipeline two requests if no effect is guaranteed to be performed after one and before the other

```
request1.map(f) *> request2
// safe to pipeline


request1.flatMap(_ => g) *> request2
// not safe to pipeline
```

Data Sources

```
trait DataSource {
  def runAll(requests: Chunk[Chunk[A]]): ZIO[R, Nothing, CompletedRequestMap]
}
```

- Data sources now take a **`Chunk[Chunk[A]]`**
- Outer chunk must be executed sequentially
- Inner chunks can be executed in parallel

Other Features

Caching

ZIO Query can automatically cache and deduplicate requests

- Extremely useful for read only queries
- Write queries in natural style without worrying about duplication
- Create consistent view of potentially changing data

Data Source Combinators

Data sources are first class values and can be composed

- **`race`** - submit requests to two data sources concurrently, returning results from the first to complete and safely interrupting the loser
- **`batchN`** - limit the maximum degree of parallelism for a data source

Data Source Aspects

```
trait DataSourceAspect[-R] {
  def apply[R1 <: R, A](dataSource: DataSource[R1, A]): DataSource[R1, A]
}
```

Transform data sources on the fly with data source aspects

- Similar to test aspects from ZIO Test
- Can modify data sources but not the types of requests they accept
- Log requests, collect metrics, or modify query execution

Effects

```
trait ZQuery[-R, +E, +A] {
  def timed: ZQuery[R with Clock, E, (Duration, A)]
}
```

Queries can be constructed from ZIO effects

- Mix and match queries with effects
- Use standard ZIO combinators to handle environment and errors
- Great for tasks such as timing query execution

Overview

- Add efficient pipelining, batching, and caching to any data source
- Optimize requests without creating a domain specific language
- Combine queries and effects using combinators you already know

Thank You

- Pierre Ricadat
- John de Goes
- Caliban users and contributors
- Courtney de Goes
- All of you