Production-ready GraphQL with Caliban

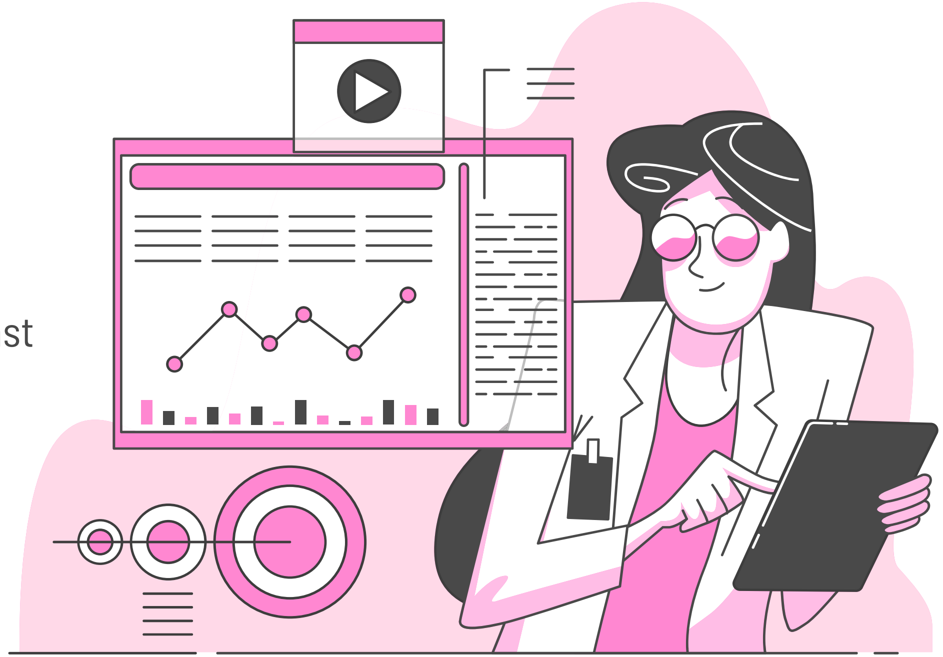
Atlanta Scala Meetup, June 2023

Pierre Ricadat

Pierre Ricadat - @ghostdogpr

Tech Lead at Devsisters
Creator of Caliban and Shardcake
OSS Contributor

Table of Contents



01

What is Caliban?

A quick primer

Caliban

GraphQL Server

Scala

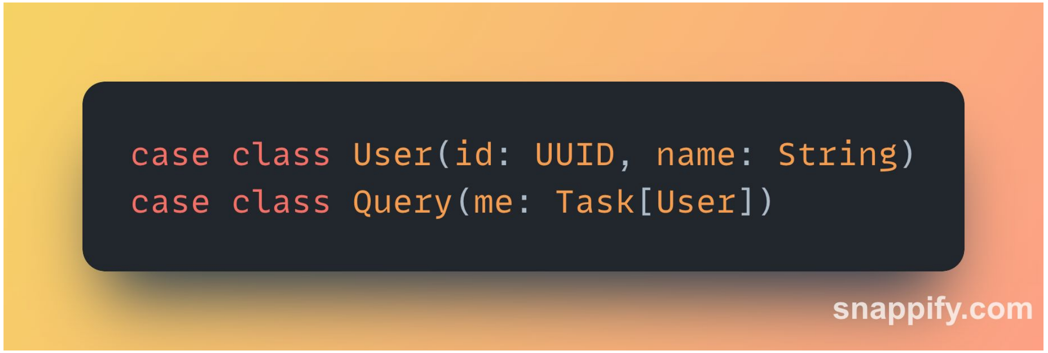
Purely functional
Minimal boilerplate
Great interop
Rich feature set

GraphQL Client

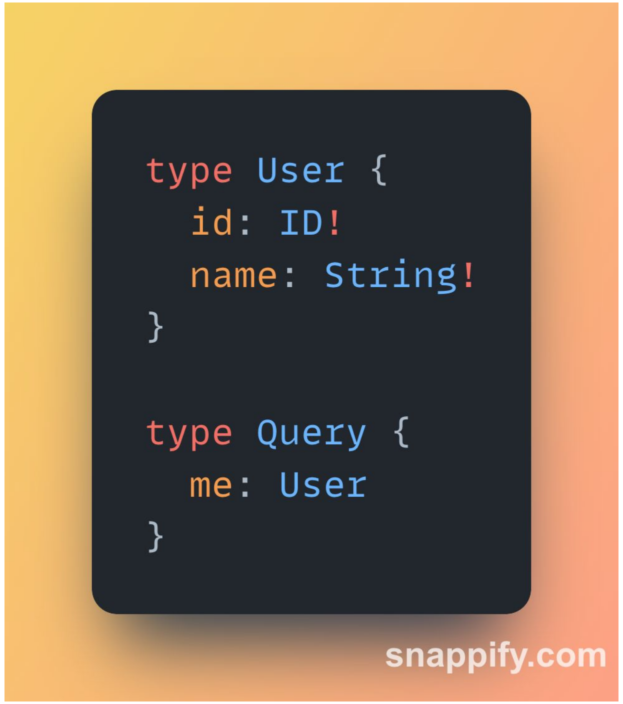
Scala
Scala.js
Scala Native

Purely functional
Type safety

Model your GraphQL schema with Scala types

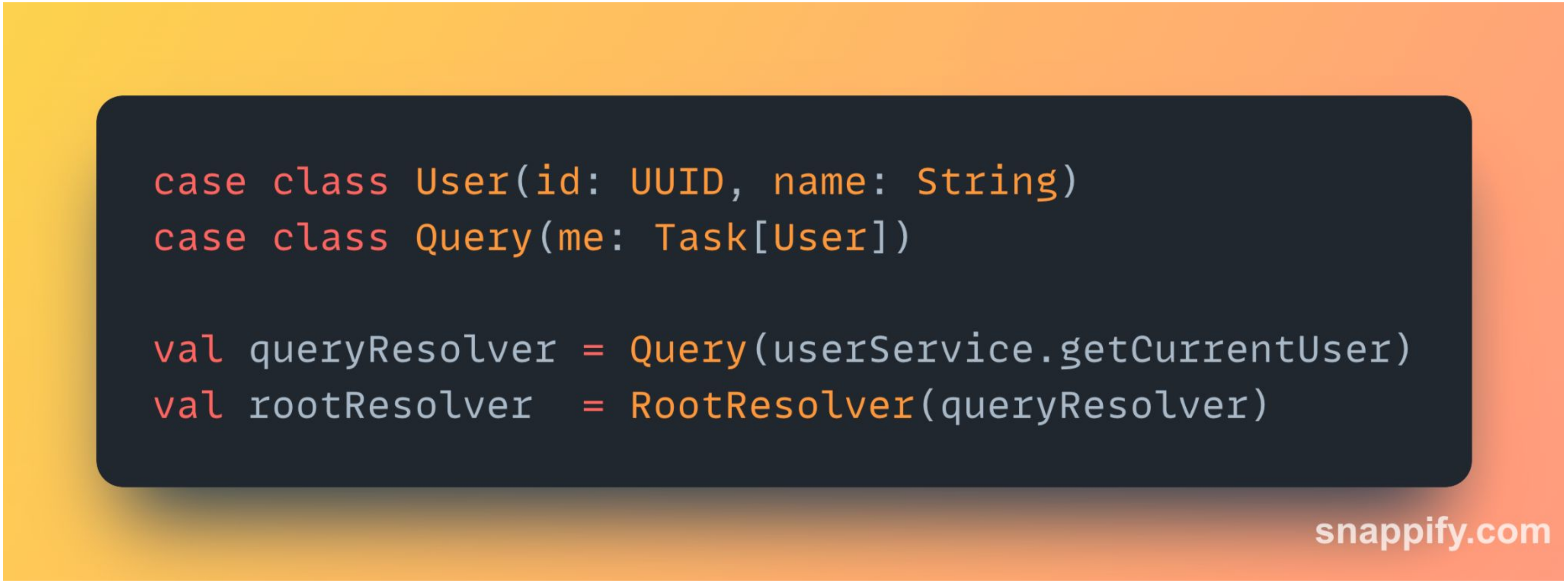


```scala
case class User(id: UUID, name: String)
case class Query(me: Task[User])
```



```graphql
type User {
  id: ID!
  name: String!
}

type Query {
  me: User
}
```

Resolver is just a value



```scala
case class User(id: UUID, name: String)
case class Query(me: Task[User])

val queryResolver = Query(userService.getCurrentUser)
val rootResolver  = RootResolver(queryResolver)
```

Scala types to GraphQL types

Supported types

Int, String, List, etc
java.util.UUID, java.time
Future, ZIO, F[_]

…

Case classes & sealed traits

Auto derivation (import)
Semi-auto derivation
(derives, given)

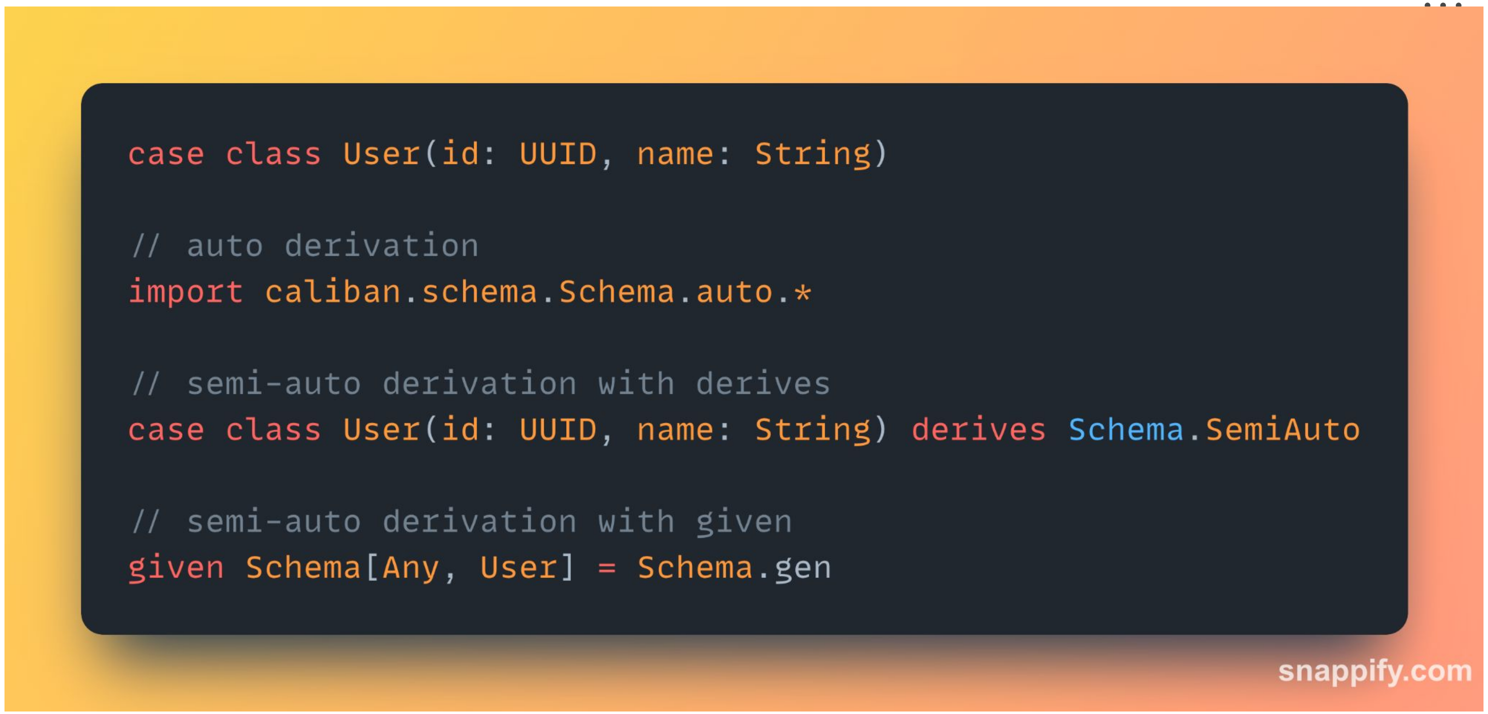
…

Other types

Custom instance of
Schema typeclass

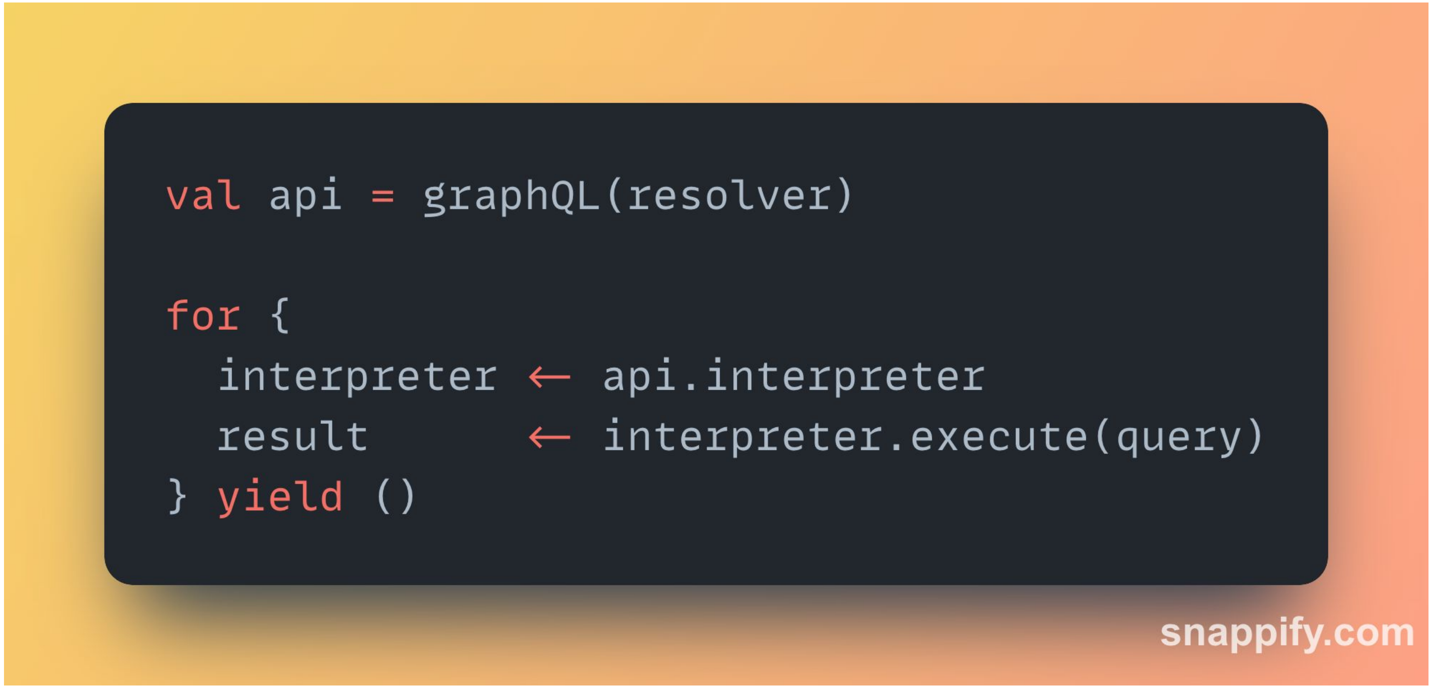
…

Derivation example



```scala
case class User(id: UUID, name: String)

// auto derivation
import caliban.schema.Schema.auto.*

// semi-auto derivation with derives
case class User(id: UUID, name: String) derives Schema.SemiAuto

// semi-auto derivation with given
given Schema[Any, User] = Schema.gen
```

From resolver to interpreter



```
val api = graphQL(resolver)

for {
  interpreter ← api.interpreter
  result      ← interpreter.execute(query)
} yield ()
```

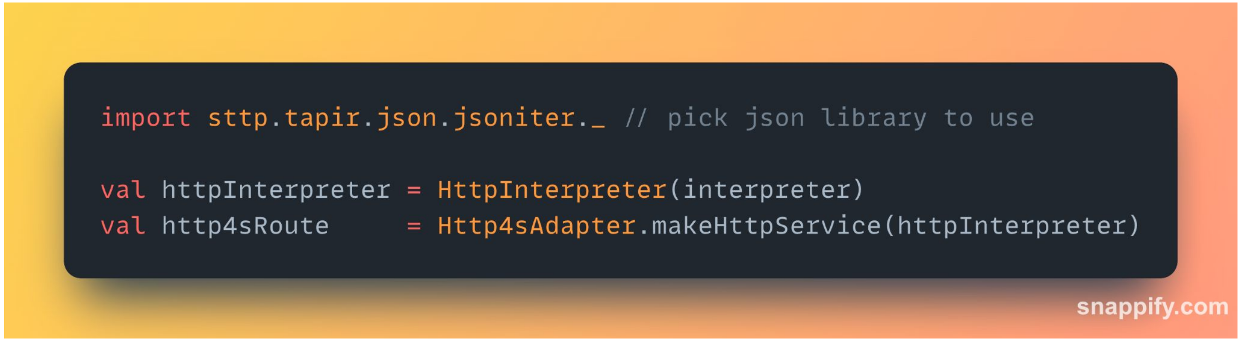
Use the http and json lib of your choice

http4s

zio http

akka http

play



```
import sttp.tapir.json.jsoniter._ // pick json library to use

val httpInterpreter = HttpInterpreter(interpreter)
val http4sRoute     = Http4sAdapter.makeHttpService(httpInterpreter)
```

02

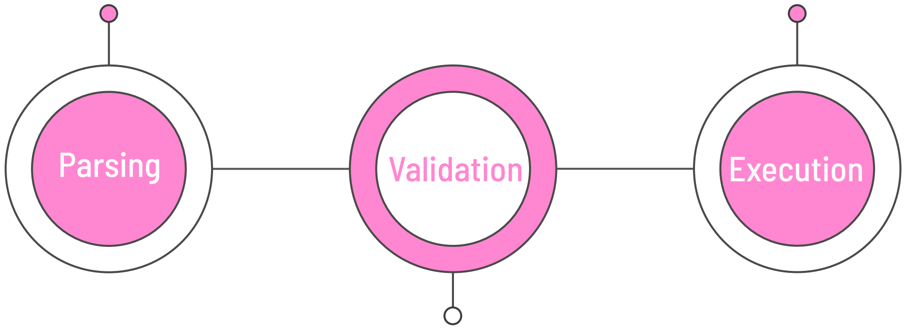
Application Monitoring

How to add logs, traces and metrics

3 phases of request processing

Transform query string into Document ADT

Run the resolvers for that query



Verify the Document ADT conforms to the spec and to the schema

Introducing wrappers

Overall Wrapper

Wrap the whole query processing

Parsing Wrapper

Wrap the parsing phase

Validation Wrapper

Wrap the validation phase

Execution Wrapper

Wrap the execution phase

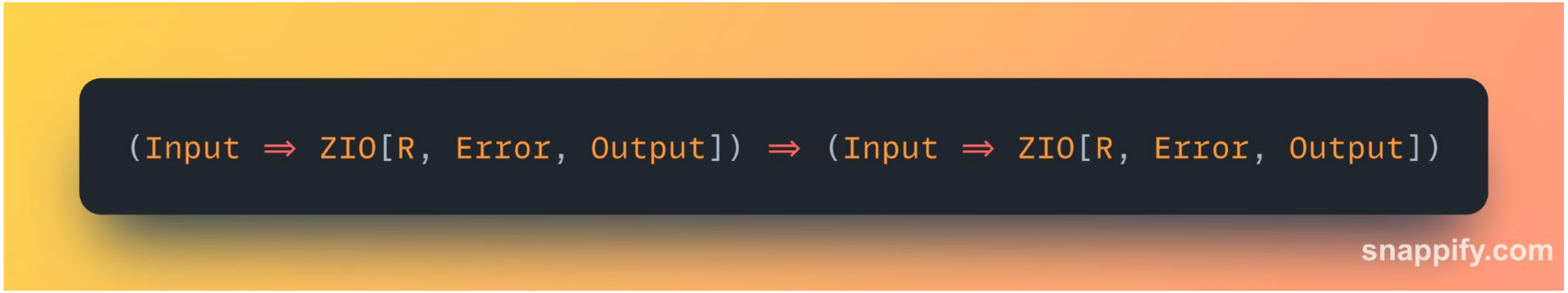
Field Wrapper

Wrap each individual field execution

Introspection Wrapper

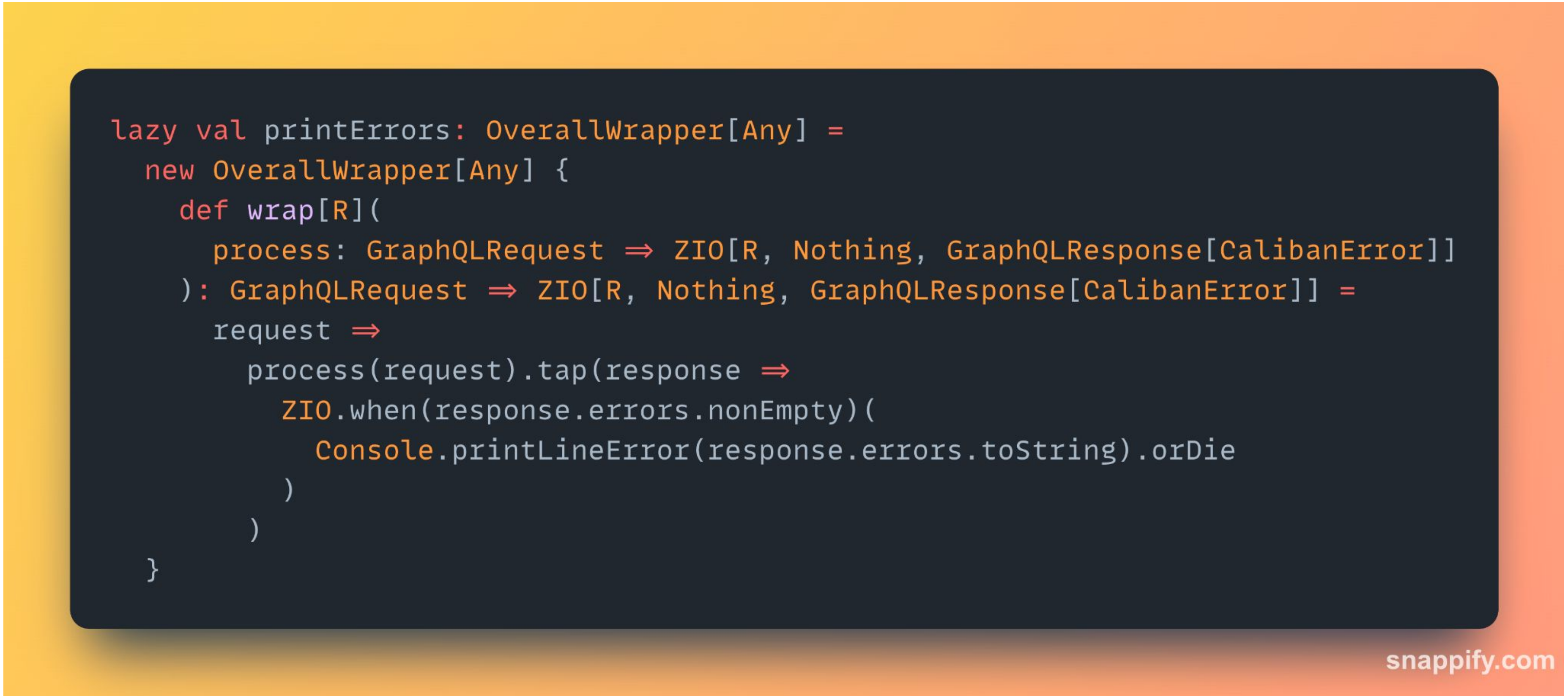
Wrap only introspection queries

What's a wrapper?



```
(Input ⇒ ZIO[R, Error, Output]) ⇒ (Input ⇒ ZIO[R, Error, Output])
```

Simple wrapper for printing errors



```scala
lazy val printErrors: OverallWrapper[Any] =
  new OverallWrapper[Any] {
    def wrap[R](
      process: GraphQLRequest ⇒ ZIO[R, Nothing, GraphQLResponse[CalibanError]]
    ): GraphQLRequest ⇒ ZIO[R, Nothing, GraphQLResponse[CalibanError]] =
      request ⇒
        process(request).tap(response ⇒
          ZIO.when(response.errors.nonEmpty)(
            Console.printLineError(response.errors.toString).orDie
          )
        )
  }
```

Pre-defined wrappers

Validation

maxDepth, maxFields, maxCost, queryCost, timeout

Logging

printErrors, printSlowQueries, logSlowQueries

Telemetry

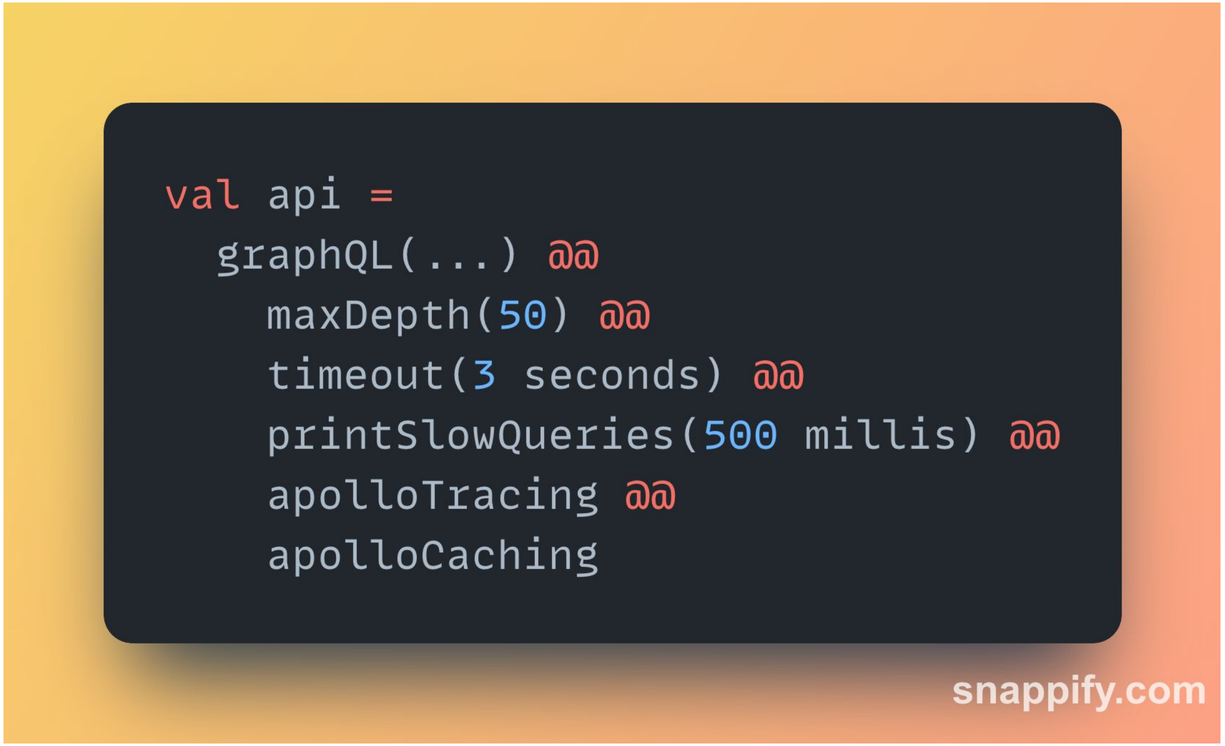
metrics, traced

Apollo Tracing

Apollo Caching

Apollo Persisted Queries

Using wrappers



```
val api =
  graphQL(...) @@
    maxDepth(50) @@
    timeout(3 seconds) @@
    printSlowQueries(500 millis) @@
    apolloTracing @@
    apolloCaching
```

03

Query Optimization

How to make your queries fast

Possible approaches

01

ZQuery

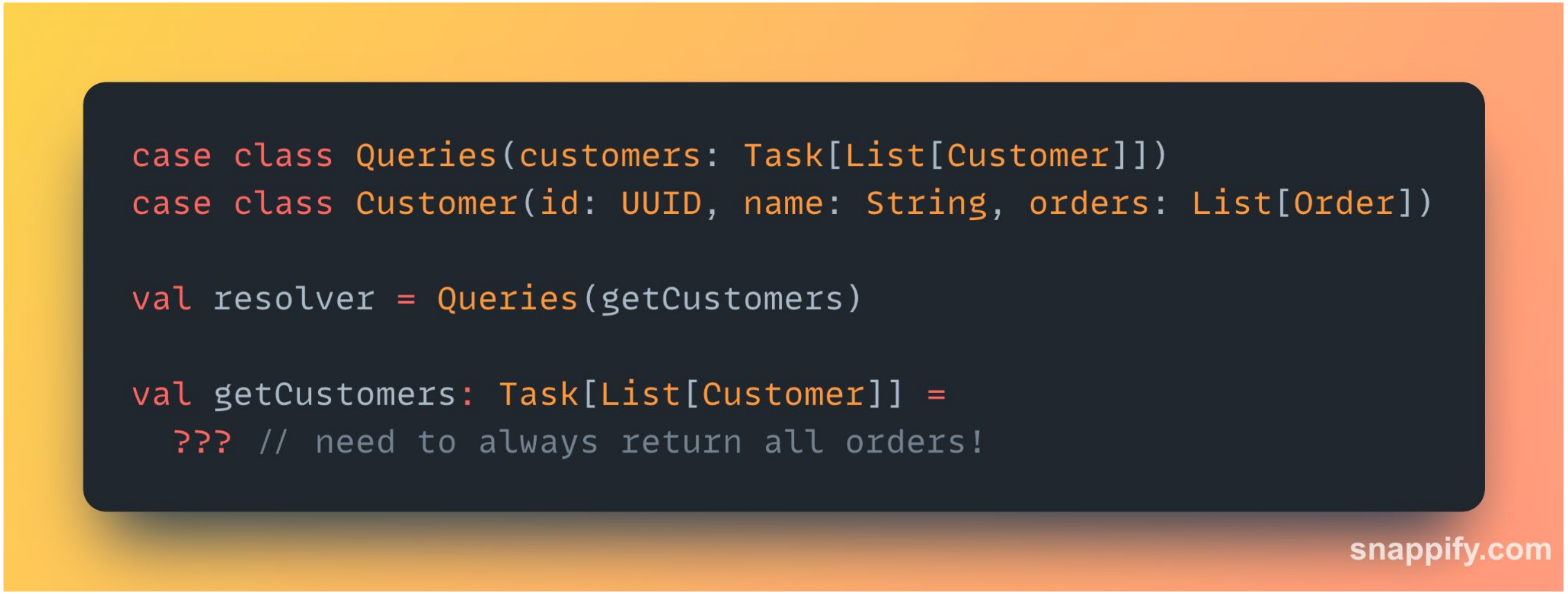
Build efficient queries despite the nesting (n+1 query problem)

02

Field metadata

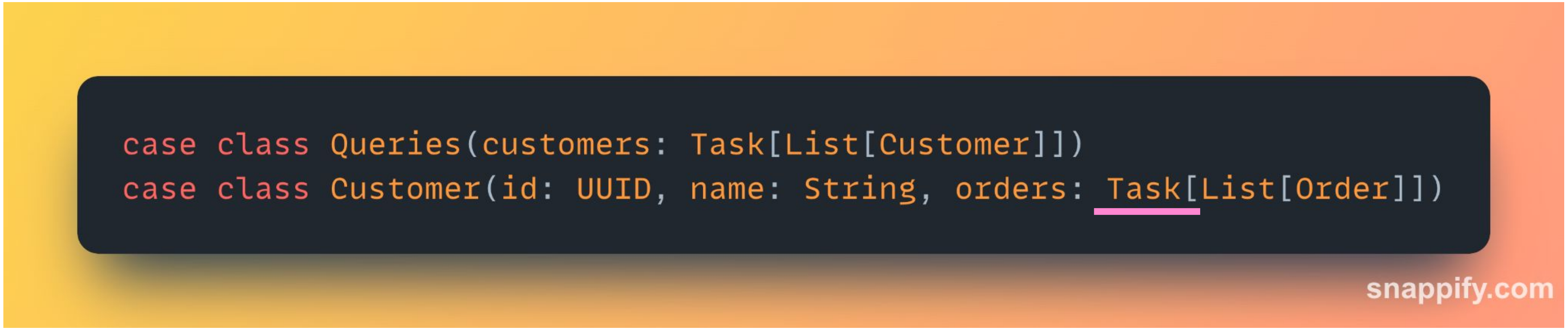
Build queries that match the client requests exactly

Do I need to query everything?



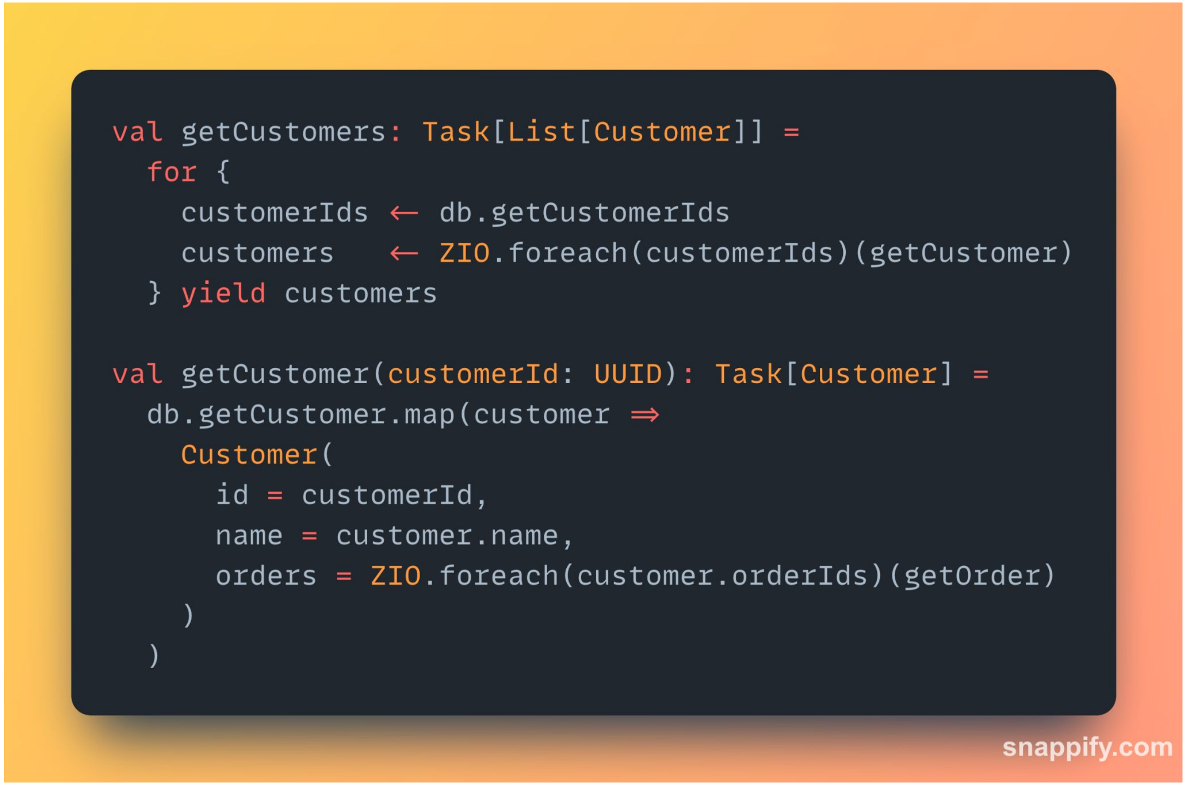
```scala
case class Queries(customers: Task[List[Customer]])
case class Customer(id: UUID, name: String, orders: List[Order])

val resolver = Queries(getCustomers)

val getCustomers: Task[List[Customer]] =
  ??? // need to always return all orders!
```

Laziness to the rescue!



```
case class Queries(customers: Task[List[Customer]])
case class Customer(id: UUID, name: String, orders: Task[List[Order]])
```

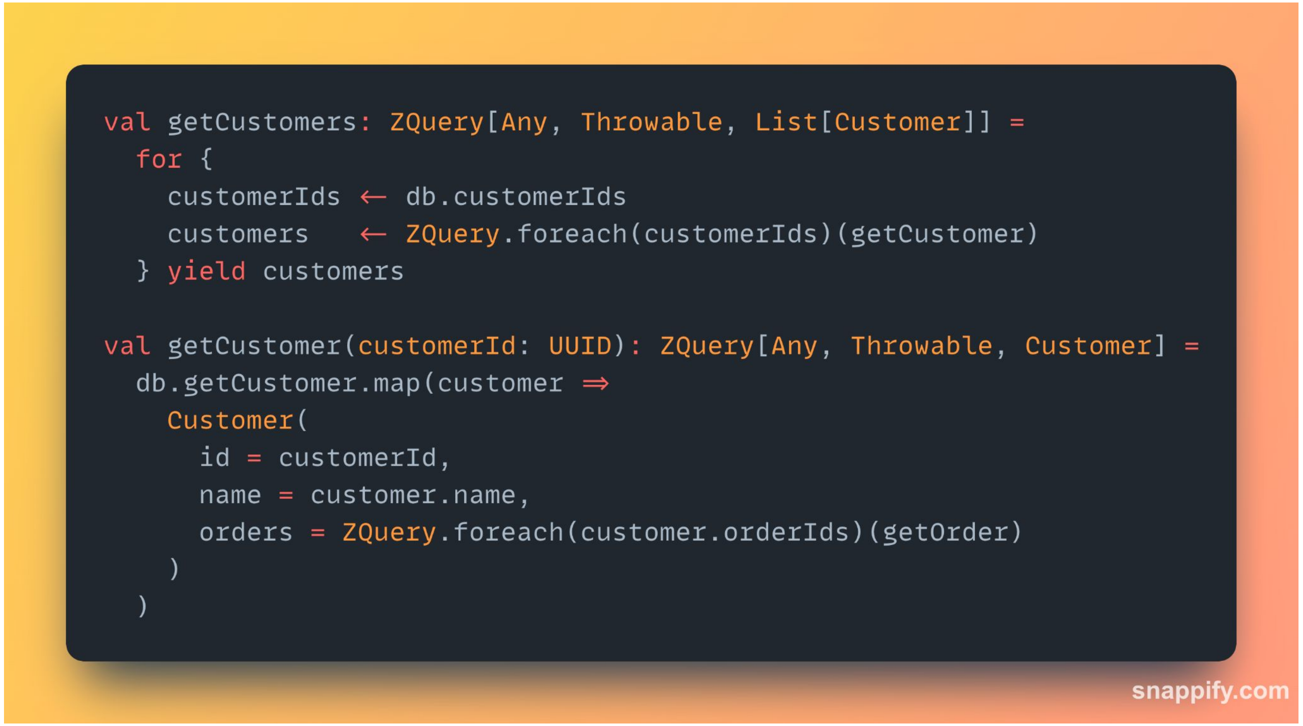
The n+1 problem



```scala
val getCustomers: Task[List[Customer]] =
  for {
    customerIds ← db.getCustomerIds
    customers   ← ZIO.foreach(customerIds)(getCustomer)
  } yield customers

val getCustomer(customerId: UUID): Task[Customer] =
  db.getCustomer.map(customer ⇒
    Customer(
      id = customerId,
      name = customer.name,
      orders = ZIO.foreach(customer.orderIds)(getOrder)
    )
  )
```

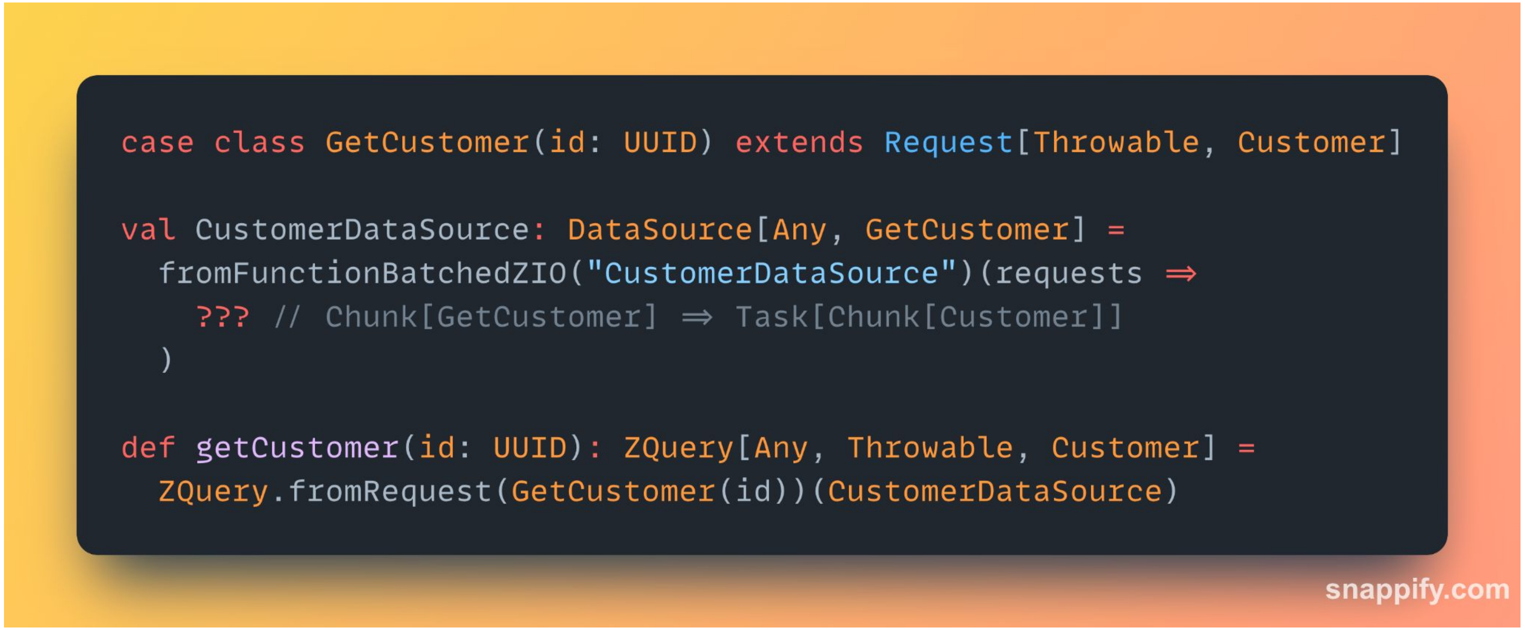
snappify.com

Replacing ZIO by ZQuery



```scala
val getCustomers: ZQuery[Any, Throwable, List[Customer]] =
  for {
    customerIds ← db.customerIds
    customers   ← ZQuery.foreach(customerIds)(getCustomer)
  } yield customers

val getCustomer(customerId: UUID): ZQuery[Any, Throwable, Customer] =
  db.getCustomer.map(customer ⇒
    Customer(
      id = customerId,
      name = customer.name,
      orders = ZQuery.foreach(customer.orderIds)(getOrder)
    )
  )
```

Defining a DataSource



```scala
case class GetCustomer(id: UUID) extends Request[Throwable, Customer]

val CustomerDataSource: DataSource[Any, GetCustomer] =
  fromFunctionBatchedZIO("CustomerDataSource")(requests ⇒
    ??? // Chunk[GetCustomer] ⇒ Task[Chunk[Customer]]
  )

def getCustomer(id: UUID): ZQuery[Any, Throwable, Customer] =
  ZQuery.fromRequest(GetCustomer(id))(CustomerDataSource)
```

What you get

Deduplication

If the same ID is requested multiple times in your GraphQL query, it will be queried from your data source only once

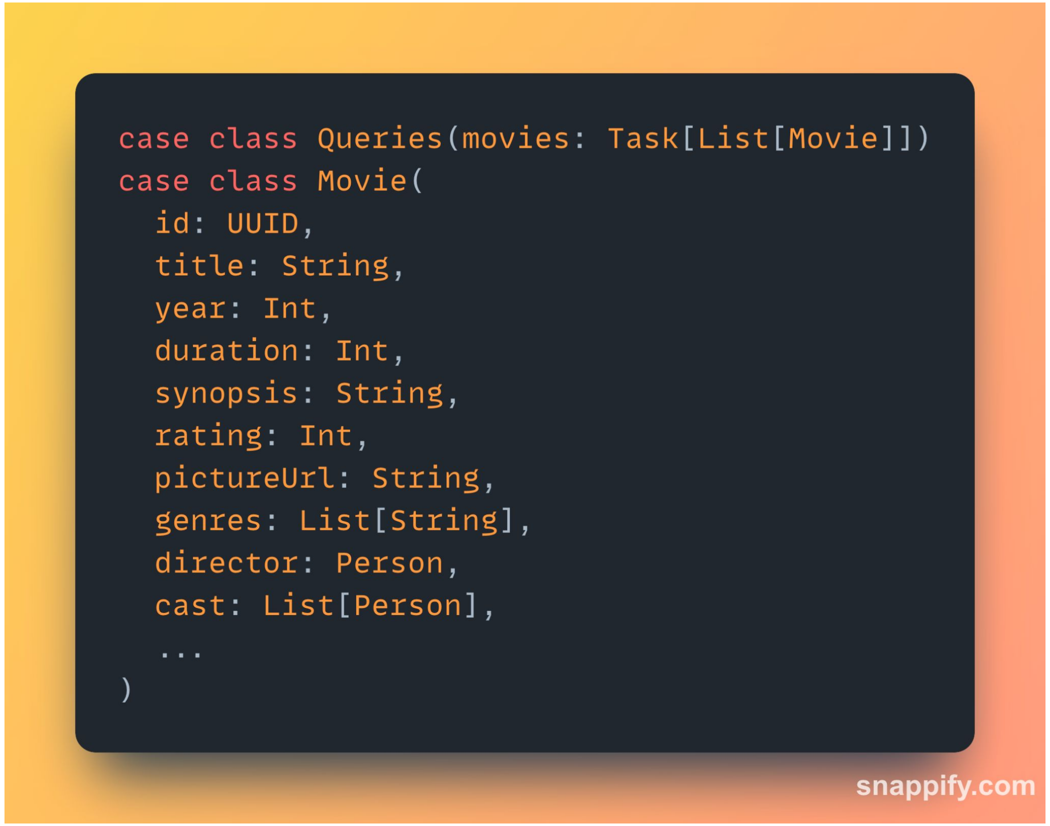
Batching

Query all the needed data at once in a single request to your data source

Parallelism Control

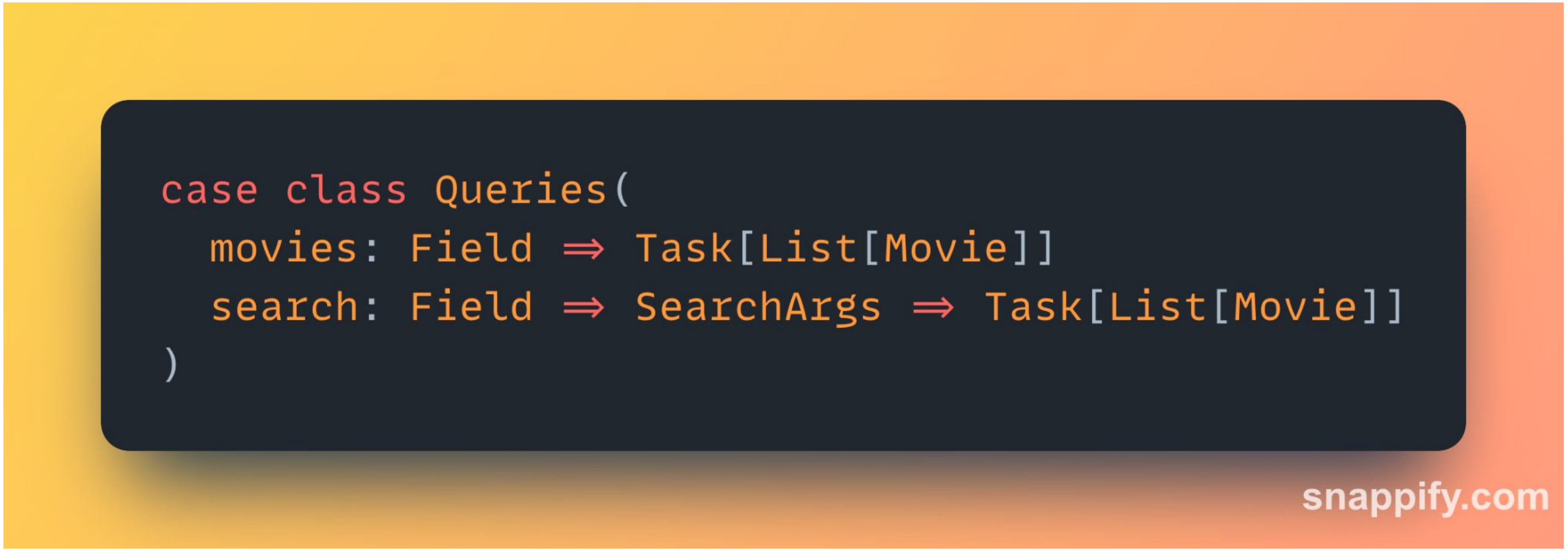
Decide what can be done in parallel and what should be done sequentially

Do I need to query all fields?



```scala
case class Queries(movies: Task[List[Movie]])
case class Movie(
  id: UUID,
  title: String,
  year: Int,
  duration: Int,
  synopsis: String,
  rating: Int,
  pictureUrl: String,
  genres: List[String],
  director: Person,
  cast: List[Person],
  ...
)
```

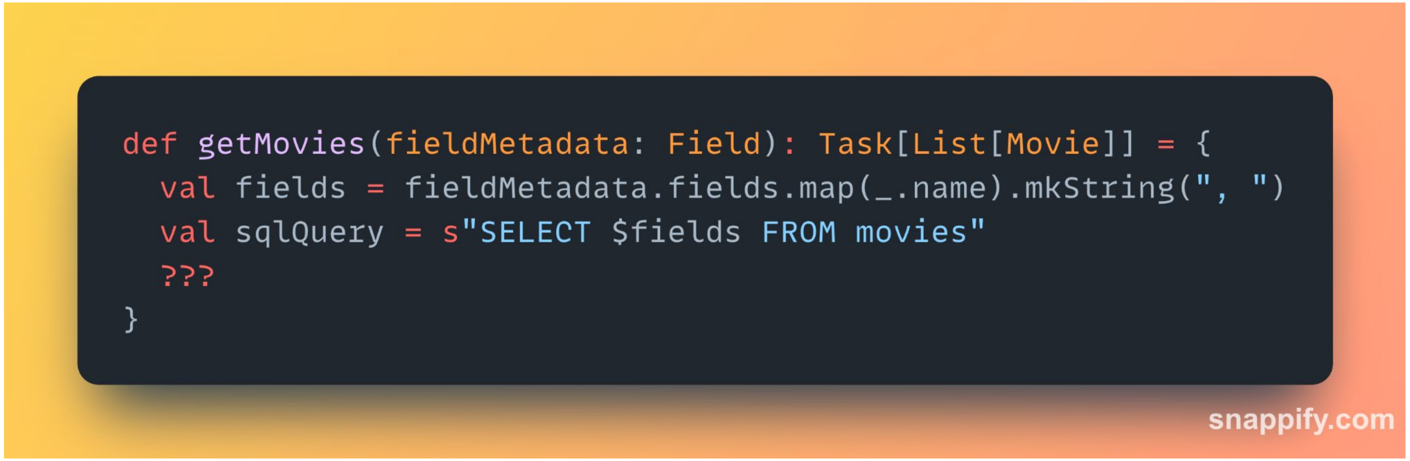
Access field metadata



```
case class Queries(
  movies: Field => Task[List[Movie]]
  search: Field => SearchArgs => Task[List[Movie]]
)
```

What's in Field?

- Field name
- Field type
- Parent type
- Inner selection <- this is particularly useful to build SQL queries
- Arguments
- Directives
- ...



```scala
def getMovies(fieldMetadata: Field): Task[List[Movie]] = {
  val fields = fieldMetadata.fields.map(_.name).mkString(", ")
  val sqlQuery = s"SELECT $fields FROM movies"
  ???
}
```

ProtoQuill Integration

https://github.com/zio/zio-protoquill#caliban-integration

```scala
// Create a query and add .filterColumns and .filterByKeys to the end
inline def peopleAndAddresses(inline columns: List[String], inline filters: Map[String, String]) =
  quote {
    // Given a query...
    query[Person].leftJoin(query[Address]).on((p, a) => p.id == a.ownerId)
      .map((p, a) => PersonAddress(p.id, p.first, p.last, p.age, a.map(_.street)))
      // Add these to the end
      .filterColumns(columns)
      .filterByKeys(filters)
  }

// Create a data-source that will pass along the column include/exclude and filter information
object DataService:
  def personAddress(columns: List[String], filters: Map[String, String]) =
    run(q(columns, filters)).provide(Has(myDataSource))
    // Assume this returns:
    // List(
    //   PersonAddress(1, "One", "A", 44, Some("123 St")),
    //   PersonAddress(2, "Two", "B", 55, Some("123 St")),
    //   PersonAddress(3, "Three", "C", 66, None),
    // )

// Create your Caliban Endpoint
case class Queries(
  personAddress: Field => (ProductArgs[PersonAddress] => Task[List[PersonAddress]])
)
```

04

Authentication

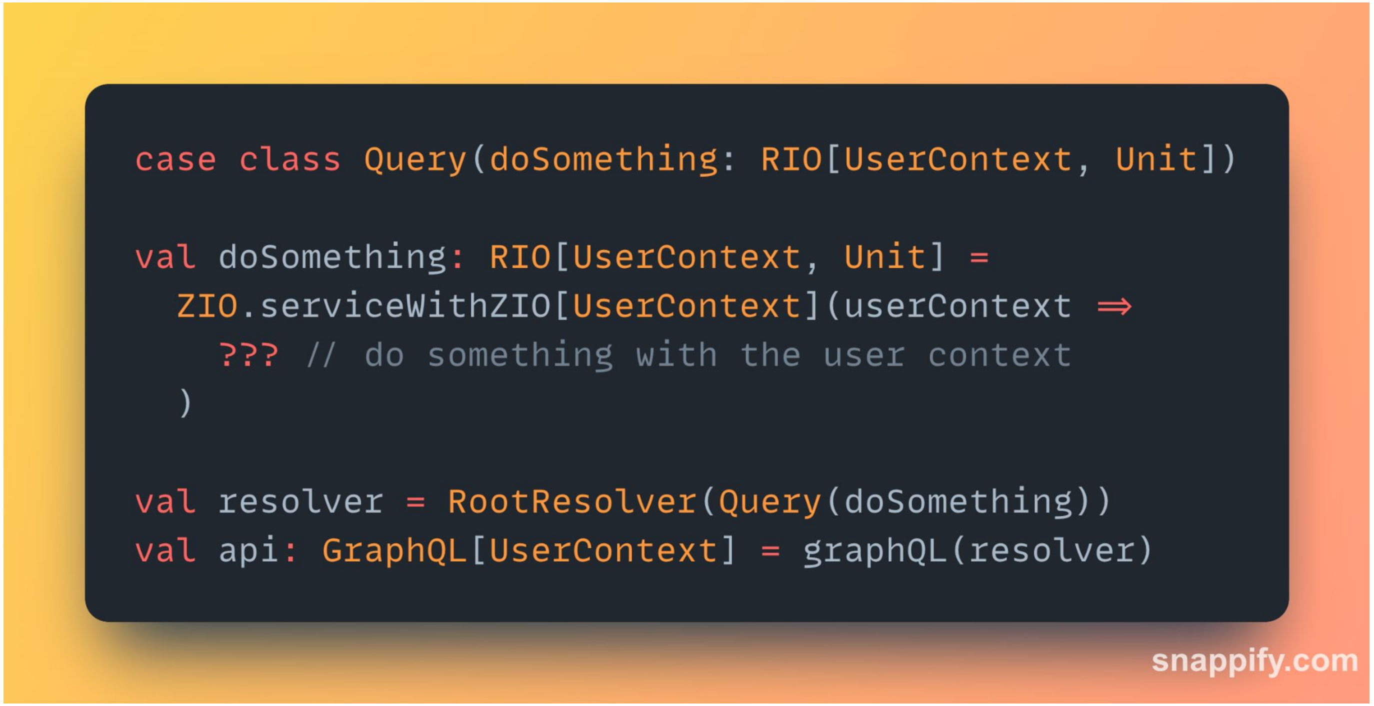
How to handle context

Authentication

Typical needs:

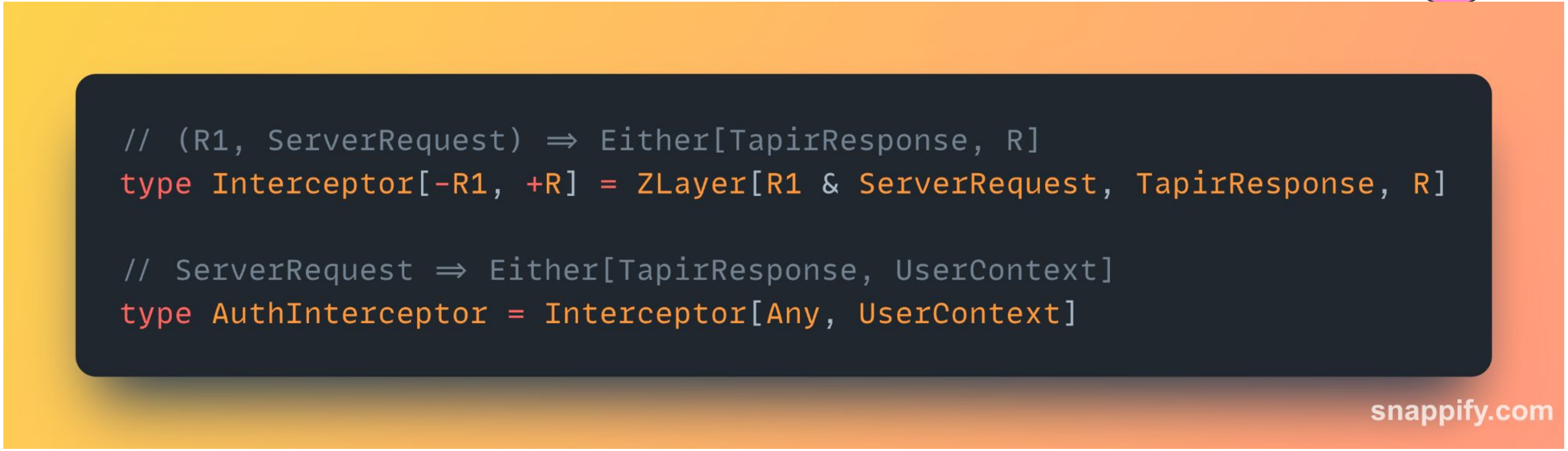
- Reject unauthenticated requests
- Access user information stored in a "context"

Use ZIO environment to store a context



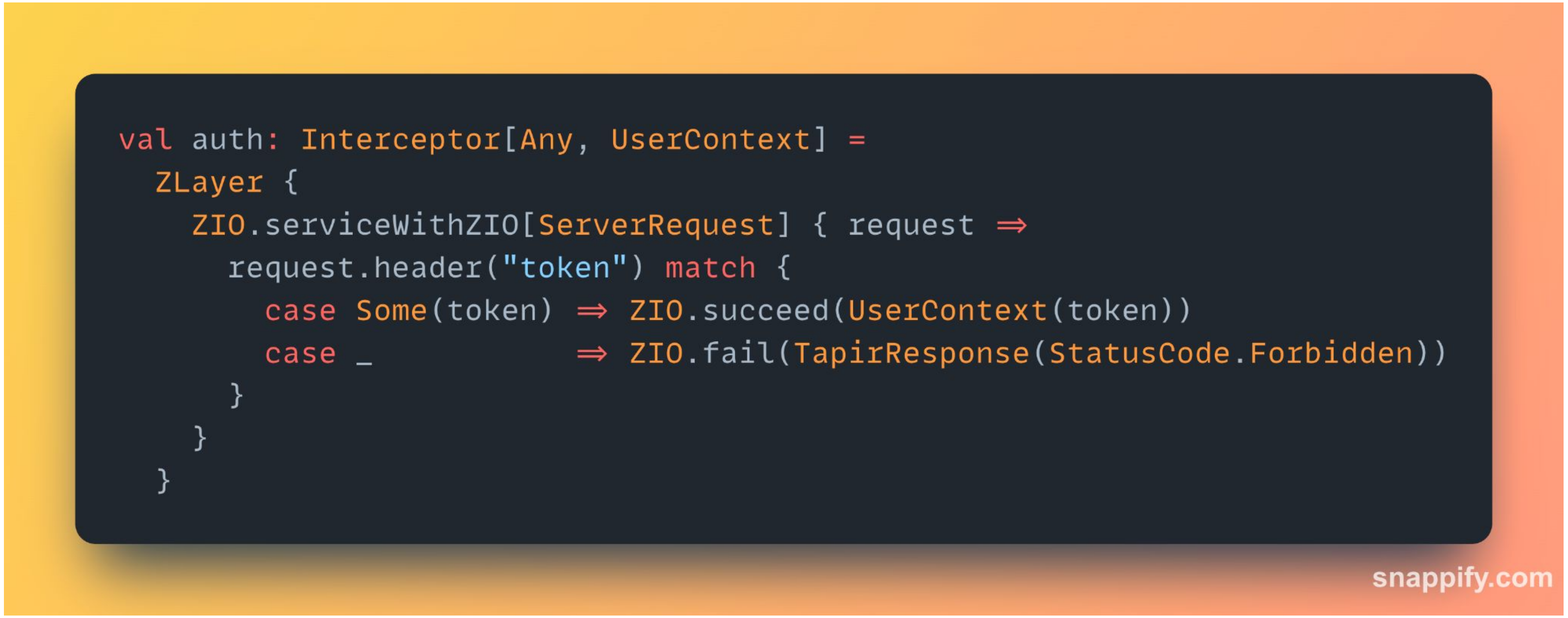
```scala
case class Query(doSomething: RIO[UserContext, Unit])

val doSomething: RIO[UserContext, Unit] =
  ZIO.serviceWithZIO[UserContext](userContext ⇒
    ??? // do something with the user context
  )

val resolver = RootResolver(Query(doSomething))
val api: GraphQL[UserContext] = graphQL(resolver)
```

Introducing Interceptors



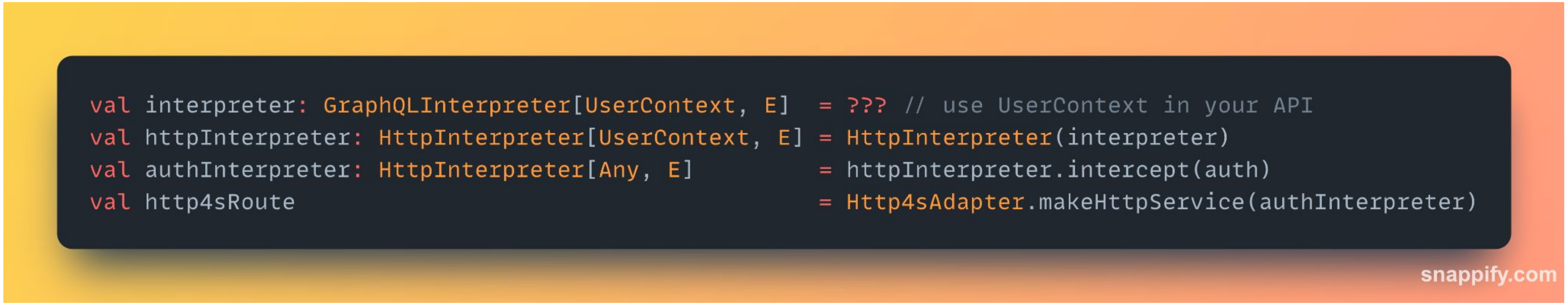
```scala
// (R1, ServerRequest) ⇒ Either[TapirResponse, R]
type Interceptor[-R1, +R] = ZLayer[R1 & ServerRequest, TapirResponse, R]

// ServerRequest ⇒ Either[TapirResponse, UserContext]
type AuthInterceptor = Interceptor[Any, UserContext]
```

A simple interceptor that gets a token from headers



```scala
val auth: Interceptor[Any, UserContext] =
  ZLayer {
    ZIO.serviceWithZIO[ServerRequest] { request =>
      request.header("token") match {
        case Some(token) => ZIO.succeed(UserContext(token))
        case _           => ZIO.fail(TapirResponse(StatusCode.Forbidden))
      }
    }
  }
```

Eliminate the environment



```
val interpreter: GraphQLInterpreter[UserContext, E]  = ??? // use UserContext in your API
val httpInterpreter: HttpInterpreter[UserContext, E] = HttpInterpreter(interpreter)
val authInterpreter: HttpInterpreter[Any, E]         = httpInterpreter.intercept(auth)
val http4sRoute                                      = Http4sAdapter.makeHttpService(authInterpreter)
```

Interceptor use cases

Authentication & Authorization

Reject wrong queries
Inject a context that can be used
in your API (and wrappers!)

Tracing

Extract tracing information from
incoming request and inject it into
the current span

Scope & Configuration

Change the value of the current
FiberRef to locally modify the
behavior of the API

There’s more!

Schema stitching and federation

Schema comparison

Client generation from server code

@defer support

Relay support

Schema reporting

Thanks!

Questions?

https://github.com/ghostdogpr/caliban

Sponsor this project

ghostdogpr Pierre Ricadat