

# Delivering GraphQL Services Using Sangria

Los Angeles Scala Users Group

Daniel Brice, CJ Affiliate ([dbrice@cj.com](mailto:dbrice@cj.com))

27 June 2018

## Designing Your GraphQL Schema

## Designing Your GraphQL Schema (1)

```
type Post {  
  id: String!  
  title: String!  
  publishedAt: DateTime!  
  likes: Int! @default(value: 0)  
  blog: Blog @relation(name: "Posts")  
}  
  
type Blog {  
  id: String!  
  name: String!  
  description: String,  
  posts: [Post!]! @relation(name: "Posts")  
}
```

Figure 1: GraphQL Schema Definition Language

## Designing Your GraphQL Schema (2)

### Database Layout

```
TABLE foo
  INT    id,
  STRING name,
  INT    bar.id
```

```
TABLE bar
  INT    id,
  STRING name
```

### GraphQL Schema

```
type Foo {
  id:    Int!
  name:  String!
  barId: Int!
}

type Bar {
  id:    Int!
  name:  String!
}

type Query {
  foos: [Foo!]!
  bars: [Bar!]!
}
```

## Designing Your GraphQL Schema (3)

### Database Layout

```
TABLE foo
  INT    id,
  STRING name,
  INT    bar.id
```

```
TABLE bar
  INT    id,
  STRING name
```

### GraphQL Schema

```
type Foo {
  id:    Int!
  name:  String!
  bar:   Bar!
}

type Bar {
  id:    Int!
  name:  String!
}

type Query {
  foos: [Foo!]!
  bars: [Bar!]!
}
```

# Designing Your GraphQL Schema (4)

## Database Layout

```
TABLE foo
  INT    id,
  STRING name,
  INT    bar.id
```

```
TABLE bar
  INT    id,
  STRING name
```

## GraphQL Schema

```
type Foo {
  id: Int!
  name: String!
  bar: Bar!
}

type Bar {
  id: Int!
  name: String!
  foos: [Foo!]!
}

type Query {
  foos: [Foo!]!
  bars: [Bar!]!
}
```

# Designing Your GraphQL Schema (5)

## Database Layout

```
TABLE foo
  INT    id,
  STRING name,
  INT    bar.id
```

```
TABLE bar
  INT    id,
  STRING name
```

## GraphQL Schema

```
type Foo {
  id: Int!
  name: String!
  bar: Bar!
}

type Bar {
  id: Int!
  name: String!
  foos: [Foo!]!
}

type Query {
  foos(ids: [Int!]): [Foo!]!
  bars(ids: [Int!]): [Bar!]!
}
```

## Executing GraphQL Queries



## Executing GraphQL Queries

```
// Global constant.
val yourSchema: Schema[YourContextType, Unit] = ...

// Usually create one per request.
val yourContext: YourContextType = ...

// Contained in POST body of incoming request.
val unparsedQuery: String = ...

// May contain a SyntaxError
val parsedQuery: Try[Document] = QueryParser.parse(unparsedQuery)

// May contain a ValidationError
val futureResult: Future[Json] = Executor.execute(
  queryAst      = parsedQuery.get, // Try.get, don't actually do this!
  userContext   = yourContext,
  schema        = yourSchema
)
```

## Defining Your Data Layer

# Defining Your Data Layer

## Database Layout

```
TABLE foo
  INT    id,
  STRING name,
  INT    bar.id
```

```
TABLE bar
  INT    id,
  STRING name
```

## Data Layer

```
case class Foo( id:    Int,
                 name:  String,
                 barId: Int    )
```

```
case class Bar( id:    Int,
                 name:  String )
```

```
trait Ctx {}
```

## Implementing Your Schema

# Implementing Your Schema (1)

## Data Layer

```
case class Foo( id:    Int,
                name:  String,
                barId: Int    )
```

```
trait Ctx {}
```

## GraphQL Schema

```
type Foo {
  id:    Int!
  name:  String!
  bar:   Bar!
}
```

## Sangria Schema Implementation

```
lazy val foo: GqlObject[Ctx, Foo] = ???
```

# Implementing Your Schema (2)

## Data Layer

```
case class Foo( id:    Int,
                name:  String,
                barId: Int    )
```

```
trait Ctx {}
```

## GraphQL Schema

```
type Foo {
  id:    Int!
  name:  String!
  bar:   Bar!
}
```

## Sangria Schema Implementation

```
lazy val foo: GqlObject[Ctx, Foo] =
  deriveObjectType[Ctx, Foo]()
```

## Implementing Your Schema (3)

### Data Layer

```
case class Foo( id:    Int,
                name:  String,
                barId: Int )
```

```
trait Ctx {}
```

### GraphQL Schema

```
type Foo {
  id:    Int!
  name:  String!
  bar:   Bar!
}
```

### Sangria Schema Implementation

```
lazy val foo: GqlObject[Ctx, Foo] =
  deriveObjectType[Ctx, Foo](
    ReplaceField(
      fieldName = "barId",
      field      = ???
    )
  )
```

## Implementing Your Schema (4)

### Data Layer

```
case class Foo( id:    Int,
                name:  String,
                barId: Int    )
```

```
trait Ctx {}
```

### GraphQL Schema

```
type Foo {
  id:    Int!
  name:  String!
  bar:   Bar!
}
```

### Sangria Schema Implementation

```
lazy val foo: GqlObject[Ctx, Foo] =
  deriveObjectType[Ctx, Foo](
    ReplaceField(
      fieldName = "barId",
      field      = GqlField(
        name      = "bar",
        fieldType = bar,
        resolve   = exe => ???
      )
    )
  )
```



## Implementing Your Schema (5)

### Data Layer

```
case class Foo( id:    Int,
                name:  String,
                barId: Int    )

trait Ctx {
  def fooBar(foo: Foo): Action[Ctx, Bar]
}
```

### GraphQL Schema

```
type Foo {
  id:    Int!
  name:  String!
  bar:   Bar!
}
```

### Sangria Schema Implementation

```
lazy val foo: GqlObject[Ctx, Foo] =
  deriveObjectType[Ctx, Foo](
    ReplaceField(
      fieldName = "barId",
      field      = GqlField(
        name      = "bar",
        fieldType = bar,
        resolve   = exe =>
          exe.ctx.fooBar(exe.value)
      )
    )
  )
```

## Implementing Your Schema (6)

### Data Layer

```
case class Bar( id:    Int,
                name: String )

trait Ctx {
  def fooBar(foo: Foo): Action[Ctx, Bar]
  def barFoos(bar: Bar): Action[Ctx, Seq[Foo]]
}
```

### GraphQL Schema

```
type Bar {
  id:    Int!
  name: String!
  foos: [Foo!]!
}
```

### Sangria Schema Implementation

```
lazy val bar: GqlObject[Ctx, Bar] =
  deriveObjectType[Ctx, Bar](
    AddFields(
      GqlField(
        name      = "foos",
        fieldType = GqlList(foo),
        resolve   = exe =>
          exe.ctx.barFoos(exe.value)
      )
    )
  )
```

## Implementing Your Schema (7)

### Data Layer

```
trait Ctx {  
  ...  
  def queryFoos()  
  def queryBars()  
}
```

### GraphQL Schema

```
type Query {  
  foos(ids: [Int!]): [Foo!]!  
  bars(ids: [Int!]): [Bar!]!  
}
```

### Sangria Schema Implementation

```
lazy val query: GqlObject[Ctx, Unit] = ???
```

## Implementing Your Schema (8)

### Data Layer

```
trait Ctx {  
  ...  
  def queryFoos(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Foo]]  
  
  def queryBars(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Bar]]  
}
```

### GraphQL Schema

```
type Query {  
  foos(ids: [Int!]): [Foo!]!  
  bars(ids: [Int!]): [Bar!]!  
}
```

### Sangria Schema Implementation

```
lazy val query: GqlObject[Ctx, Unit] = ???
```

# Implementing Your Schema (9)

## Data Layer

```
trait Ctx {  
  ...  
  def queryFoos(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Foo]]  
  
  def queryBars(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Bar]]  
}
```

## GraphQL Schema

```
type Query {  
  foos(ids: [Int!]): [Foo!]!  
  bars(ids: [Int!]): [Bar!]!  
}
```

## Sangria Schema Implementation

```
lazy val ids:  
  GqlArgument[Option[Seq[Int]]] =  
    GqlArgument(  
      name          = "ids",  
      argumentType =  
        GqlOptionInput(GqlListInput(GqlInt  
    ))  
  
lazy val query: GqlObject[Ctx, Unit] = ???
```

## Implementing Your Schema (10)

### Data Layer

```
trait Ctx {  
  ...  
  def queryFoos(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Foo]]  
  
  def queryBars(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Bar]]  
}
```

### GraphQL Schema

```
type Query {  
  foos(ids: [Int!]): [Foo!]!  
  bars(ids: [Int!]): [Bar!]!  
}
```

### Sangria Schema Implementation

```
lazy val ids = ...  
  
lazy val query: GqlObject[Ctx, Unit] =  
  GqlObject(  
    name    = "Query",  
    fields = gqlFields[Ctx, Unit](  
      ???, // foos field  
      ??? // bars field  
    )  
  )
```

# Implementing Your Schema (11)

## Data Layer

```
trait Ctx {  
  ...  
  def queryFoos(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Foo]]  
  
  def queryBars(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Bar]]  
}
```

## GraphQL Schema

```
type Query {  
  foos(ids: [Int!]): [Foo!]!  
  bars(ids: [Int!]): [Bar!]!  
}
```

## Sangria Schema Implementation

```
lazy val query: GqlObject[Ctx, Unit] =  
  GqlObject(  
    name      = "Query",  
    fields = gqlFields[Ctx, Unit](  
      GqlField(  
        name      = "foos",  
        fieldType = GqlList(foo),  
        arguments = List(ids),  
        resolve   = exe => ???  
      ),  
      GqlField(  
        name      = "bars",  
        fieldType = GqlList(bar),  
        arguments = List(ids),  
        resolve   = exe => ???  
      )  
    )  
  )  
)
```

## Implementing Your Schema (12)

### Data Layer

```
trait Ctx {  
  ...  
  def queryFoos(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Foo]]  
  
  def queryBars(  
    ids: Option[Seq[Int]]  
  ): Action[Ctx, Seq[Bar]]  
}
```

### GraphQL Schema

```
type Query {  
  foos(ids: [Int!]!): [Foo!]!  
  bars(ids: [Int!]!): [Bar!]!  
}
```

### Sangria Schema Implementation

```
lazy val query: GqlObject[Ctx, Unit] =  
  GqlObject(  
    name      = "Query",  
    fields = gqlFields[Ctx, Unit](  
      GqlField(  
        ...  
        resolve = exe =>  
          exe.ctx.queryFoos(exe.arg(ids))  
      ),  
      GqlField(  
        ...  
        resolve = exe =>  
          exe.ctx.queryBars(exe.arg(ids))  
      )  
    )  
  )
```