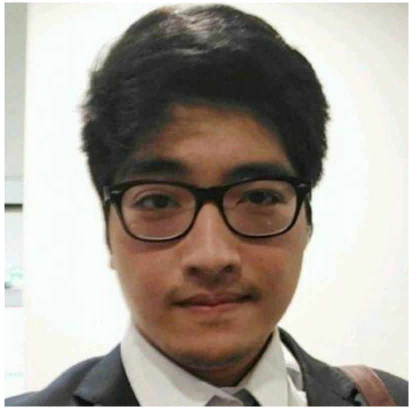
GraphQL with Sangria

Scala Implementation

About Me

Anthony SSI YAN KAI

- Data Engineer at **ebiz**next
    - Formation BI
    - **DataLake** (creation, enriching)
    - **FullStack** (react.js - scala)



Scala - Spark - Hadoop - ElasticSearch - Kafka - Akka - GraphQL

What is GraphQL all about ?

GraphQL is a **query language** for APIs

Created to answer those problematics :

- Mobile usage:
  - Sloppy networks and low-powered devices
  - Reduce the amount of data transferred over network
- Large variety of clients:
  - Classic API can hardly fit every clients needs in a maintainable way
  - Each client accesses the data it needs with a single endpoint
- Fast development
  - Classic REST APIs regularly needs to update the way data is exposed (multiple endpoints, etc…)
  - Less communication between frontend and backend

REST vs GraphQL

The **major paradigm variation**:

REST imposes a fixed data structure to the client

while …

GraphQL allow clients to fetch the data they need

Instead of multiple endpoints that returns fixed data structures, GraphQL provides an unique endpoint that returns the data asked by the client

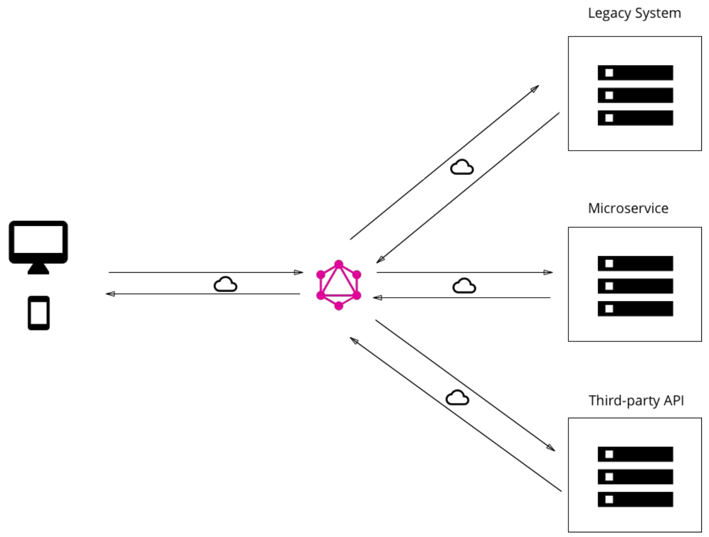
Fetching data with GraphQL can be done in a single query while in REST multiples queries are often needed and more unnecessary data is transferred.

For more GraphQL vs REST: www.howtographql.com

Architecture

GraphQL can be plugged to basically anything ...

- Database
- Legacy systems
- Microservice
- Third-party API
- Anything ...



resources: www.howtographql.com

Use Case

Expose data for a movie application

case class Movie

```
id: String
title: String,
description: Option[String]
genres: Seq[Genre.Value]
actors: Seq[Actor]
rating: Option[Double]
reviews: Seq[Review]
budget: Long
gross: Seq[Long]
```

case class Actor

```
id: String,
fullname: String
popularity: Option[Int]
movies: Seq[Movie]
```

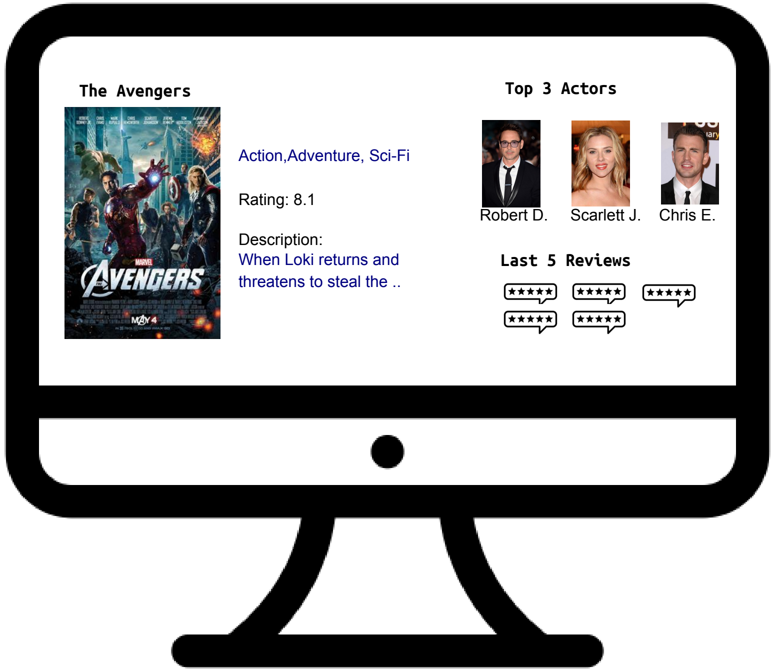
case class Review

```
id: String = UUID.randomUUID().toString,
author: String,
content: String,
rating: Double,
created: Long = Instant.now.toEpochMilli
```

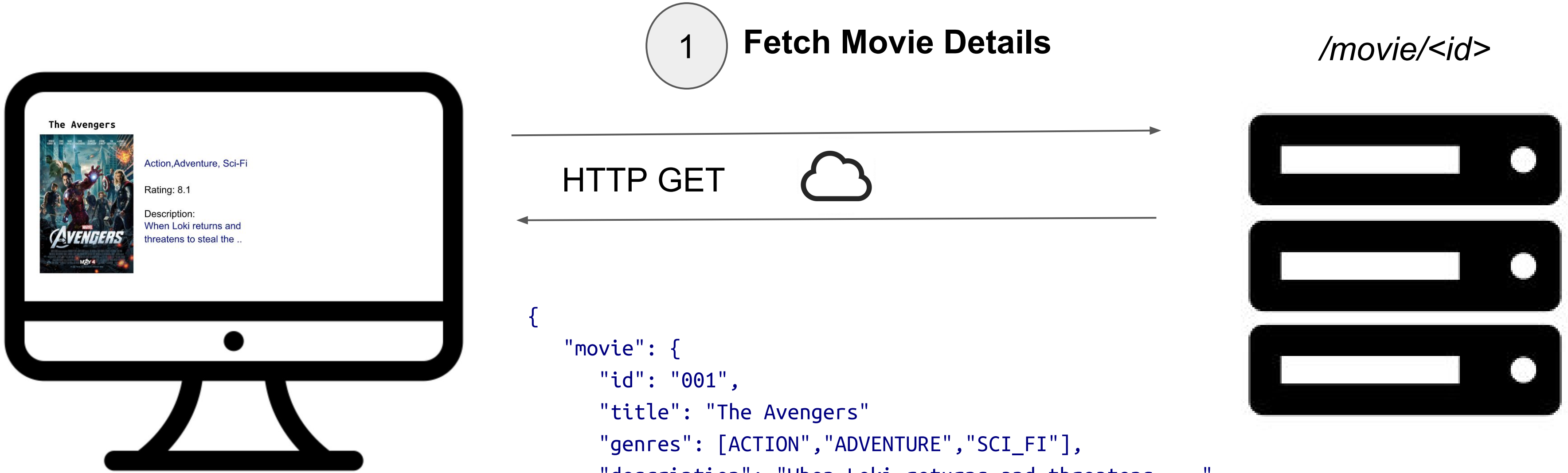
Scenario

We want to fetch this data:

- Movie Title - Genres - Rating - Description
- Top 3 most popular actors in the movie
- Last 5 reviews about the movie

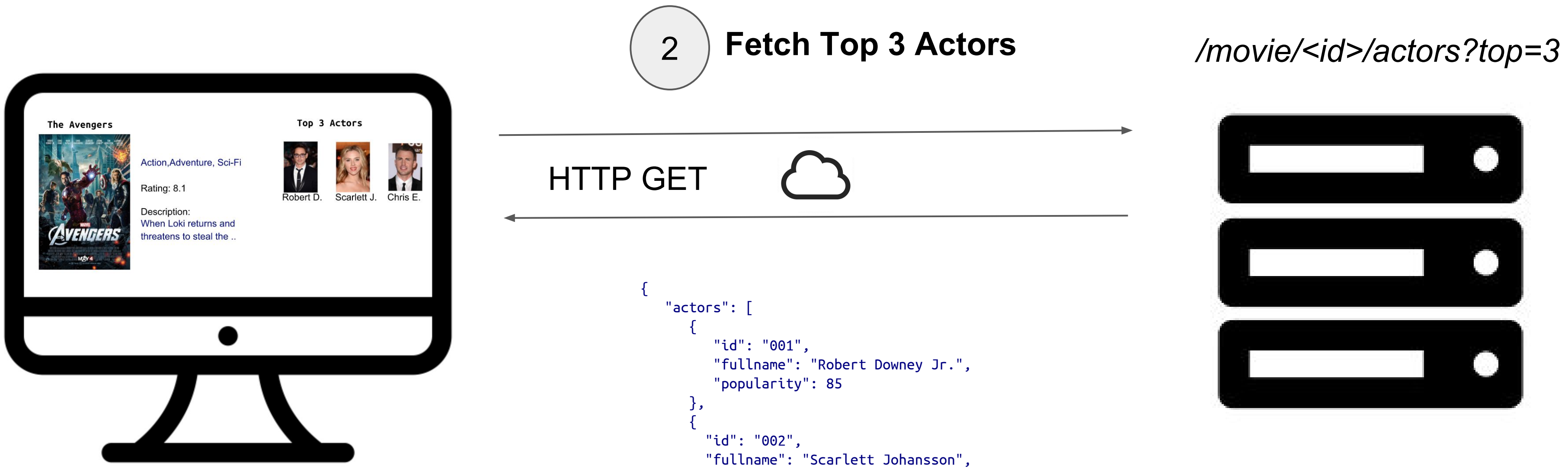


Scenario with REST



```
{
   "movie": {
      "id": "001",
      "title": "The Avengers"
      "genres": [ACTION","ADVENTURE","SCI_FI"],
      "description": "When Loki returns and threatens ...",
      "gross": [207438708,103052274,55644102,36686871],
      "rating": 8.1,
      "budget": 225000000
    }
}
```

Scenario with REST

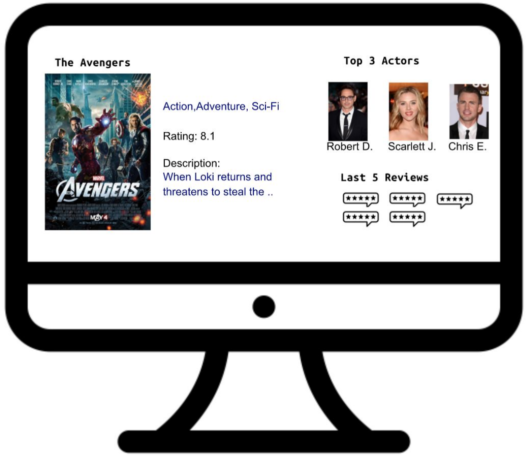


```
{
   "actors": [
      {
         "id": "001",
         "fullname": "Robert Downey Jr.",
         "popularity": 85
      },
      {
        "id": "002",
        "fullname": "Scarlett Johansson",
        "popularity": 78
      },
      {
        "id": "003",
        "fullname": "Chris Evans",
        "popularity": 78
      }
   ]
}
```

Scenario with REST

3 **Fetch Movie Reviews**

/movie/<id>/reviews?last=5

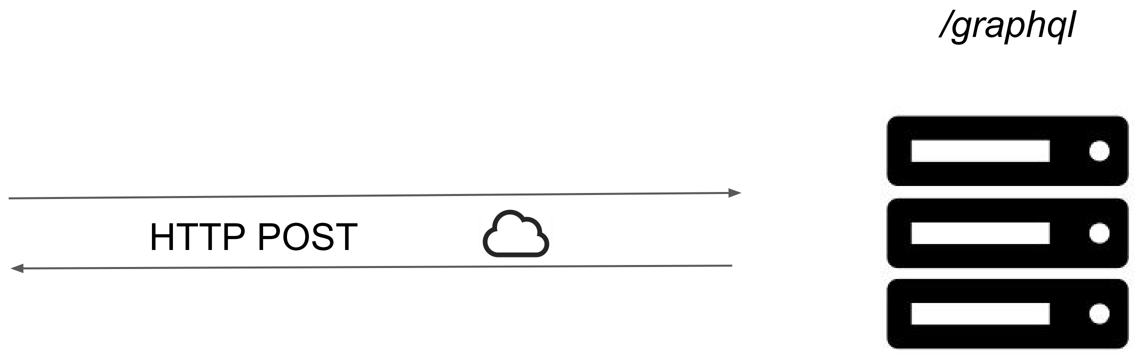


HTTP GET

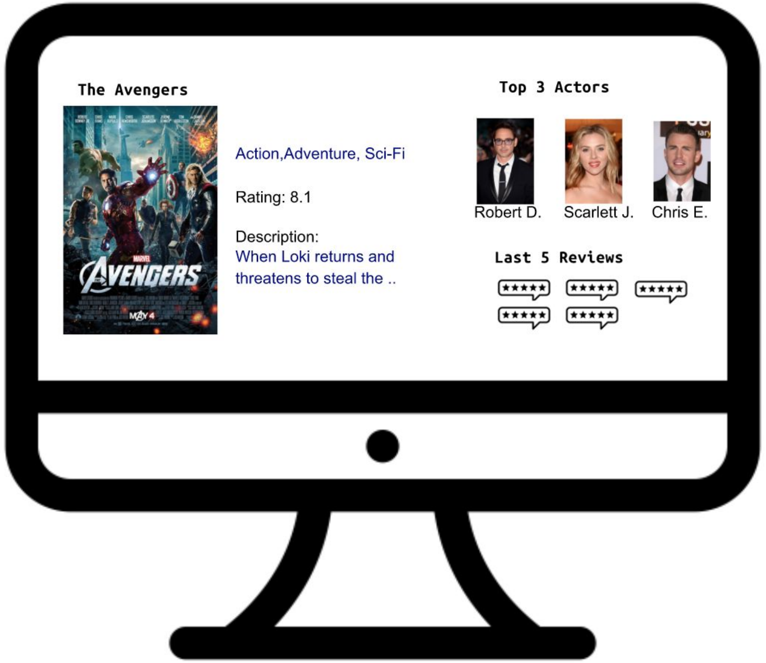
```
{
   "reviews": [
      {
         "id": "a393dc48-0bc9-4d42-bfa2-c29bb23d0cb3",
         "author": "Alex",
         "rating": 10,
         "content": "Amazing movie ! I love superheroes"
      },
      {
         "id": "010b3dbd-d24d-4349-999b-27db1135a839",
         "author": "Hannah",
         "rating": 7.5,
         "content": "Best Marvel ever."
      },
      {
         "id": "786d22dc-821e-4843-a58d-210ba097158b",
         "author": "David",
         “rating": 7,
         "content": "I didn't have high expectations and wasn't disappointed
       },
      ...
   ]
}
```

Scenario with GraphQL

```graphql
query {
  movie(id: "001") {
    title
    description
    genres
    topActors(top: 3) {
      fullname
    }
    lastReviews(last: 5) {
      author
      rating
      content
    }
  }
}
```



Scenario with GraphQL

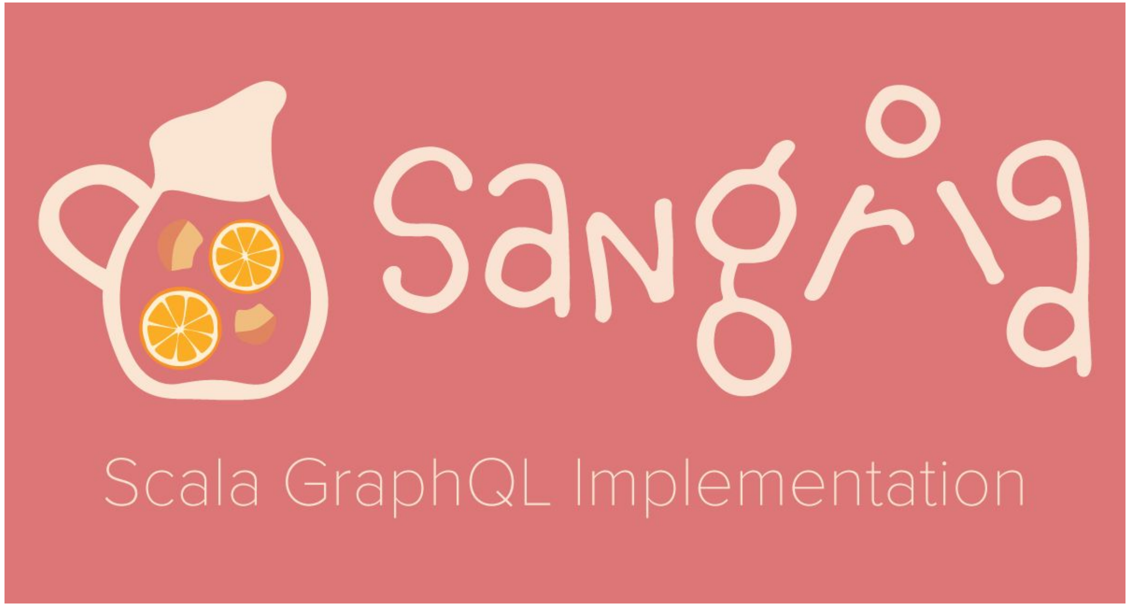


```
{
  "data": {
    "movie": {
      "title": "The Avengers",
      "description": "When Loki returns and threatens
...",
      "genres": [
        "ACTION",
        "ADVENTURE",
        "SCI_FI"
      ],
      "lastReviews": [
        {
          "author": "Alex",
          "rating": 10,
          "content": "Amazing movie ! I love superheroes"
        },
        ...
      ],
      "topActors": [
        {
          "fullname": "Robert Downey Jr.",
        },
        ...
      ]
    }
  }
}
```

When Scala met GraphQL: Sangria

Scala uses a type system … and so does GraphQL

Sangria will help us implement GraphQL in Scala



Oleg LLYENKO

GraphQL and Scala Type Systems

Trait - Interface

```scala
 trait VideoContext {
 val id: String
 val title: String
 val description: Option[String]
 val genres: Seq[Genre.Value]
 val actors: Seq[Actor]
 val rating: Option[Double]
 val reviews: Seq[Review]

 @GraphQLField
 def topActors(top: Int): Seq[Actor] =
actors.sortBy(_.popularity).reverse.take(top)

 @GraphQLField
 def lastReviews(last: Int): Seq[Review] =
reviews.takeRight(last)
}
```

```graphql
interface VideoContext {
  id: String!
  title: String!
  description: String
  genres: [Genre!]!
  actors: [Actor!]!
  rating: Float
  reviews: [Review!]!
  topActors(top: Int!): [Actor!]!
  lastReviews(last: Int!): [Review!]!
}
```

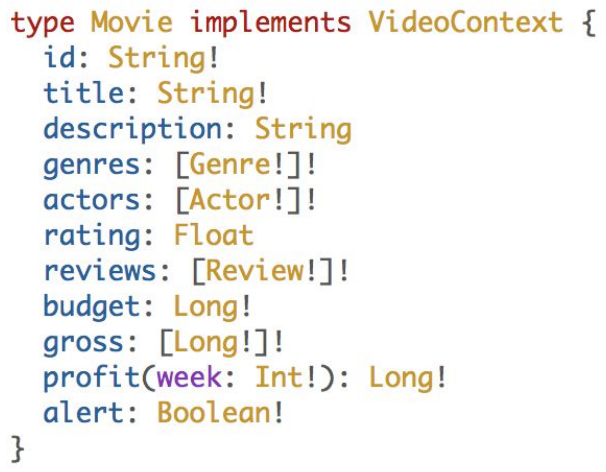
Case Class - Type

```scala
case class Movie(id: String,
                 title: String,
                 description: Option[String],
                 genres: Seq[Genre.Value],
                 actors: Seq[Actor],
                 rating: Option[Double],
                 reviews: Seq[Review],
                 budget: Long,
                 gross: Seq[Long]
                ) extends VideoContext {

 @GraphQLField
 def profit(week: Int): Long = gross.take(week - 1).sum - budget

 @GraphQLField
 def alert: Boolean = gross.sum < 0

}
```



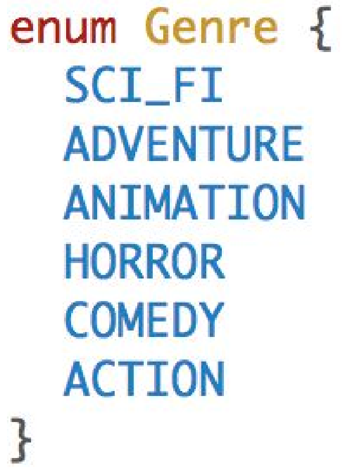
```graphql
type Movie implements VideoContext {
  id: String!
  title: String!
  description: String
  genres: [Genre!]!
  actors: [Actor!]!
  rating: Float
  reviews: [Review!]!
  budget: Long!
  gross: [Long!]!
  profit(week: Int!): Long!
  alert: Boolean!
}
```

Enumeration - Enum

```scala
object Genre extends Enumeration {
 val ACTION, COMEDY, HORROR, ANIMATION, ADVENTURE, SCI_FI = Value
}
```

```scala
sealed trait GenreEnum

object GenreEnum {
 case object ACTION extends GenreEnum
 case object COMEDY extends GenreEnum
 case object HORROR extends GenreEnum
 case object ANIMATION extends GenreEnum
 case object ADVENTURE extends GenreEnum
 case object SCI_FI extends GenreEnum
}
```



```java
enum Genre {
   SCI_FI
   ADVENTURE
   ANIMATION
   HORROR
   COMEDY
   ACTION
}
```

Schema Definition in Sangria

Interface Type - Sangria

```
val VideoType: InterfaceType[Unit, VideoContext] = InterfaceType(
 name = "VideoContext",
 description = "Template of any video",
 fields = fields[Unit, VideoContext](
   Field(name = "id", fieldType = StringType, resolve = _.value.id),
   Field(name = "title", fieldType = StringType, resolve = _.value.title),
   Field(name = "description", fieldType = OptionType(StringType), resolve = _.value.description),
   Field(name = "genres", fieldType = ListType(GenreEnum), resolve = _.value.genres),
   Field(name = "actors", fieldType = ListType(ActorType), resolve = _.value.actors),
   Field(name = "rating", fieldType = OptionType(FloatType), resolve = _.value.rating),
   Field(name = "topActors", ListType(ActorType), arguments = Argument("top", IntType) :: Nil,
     resolve = c => c.value.topActors(c.arg(Argument("top", IntType)))),
   Field(name = "lastReviews", ListType(ReviewType), arguments = Argument("last", IntType) :: Nil,
     resolve = c => c.value.lastReviews(c.arg(Argument("last", IntType))))
 ))
```

Object Type - Sangria

```scala
val MovieType = ObjectType(
 "Movie",
 "VideoContext you can watch at the theater",
 interfaces[Unit, Movie](VideoType),
 fields = fields[Unit, Movie](
   Field("id", StringType, Some("the id of the movie."), resolve = _.value.id),
   Field("title", StringType, Some("the title of the movie."), resolve = _.value.title),
   Field("actors", ListType(ActorType), Some("the actors appearing in the movie"), resolve = _.value.actors),
   Field("genres", ListType(GenreEnum), Some("the genres of the movie"), resolve = _.value.genres),
   Field("rating", OptionType(FloatType), resolve = _.value.rating),
   Field("reviews", ListType(ReviewType), resolve = _.value.reviews),
   Field("budget", LongType, resolve = _.value.budget),
   Field("gross", ListType(LongType), resolve = _.value.gross),
   Field("profit", LongType, arguments = Argument("week", IntType) :: Nil,
     resolve = ctx => ctx.value.profit(ctx.arg(Argument("week", IntType)))),
   Field("alert", BooleanType, resolve = _.value.alert)
 ))
```

Enum Type - Sangria

```
val GenreEnum = EnumType(
 "Genre",
 Some("Genre of a video"),
 List(
   EnumValue("COMEDY", value = Genre.COMEDY, description = Some("Toc Toc Toc ...")),
   EnumValue("ACTION", value = Genre.ACTION, description = Some("Action videos")),
   EnumValue("HORROR", value = Genre.HORROR, description = Some("Bouuuh !")),
   EnumValue("ANIMATION", value = Genre.ANIMATION, description = Some("Perfect to have a good time")),
   EnumValue("ADVENTURE", value = Genre.ADVENTURE, description = Some("Adventure videos")),
   EnumValue("SCI_FI", value = Genre.SCI_FI, description = Some("Weird stuff happening"))
 )
)
```

If Scala and GraphQL have such similar type systems
Why do we have to redefine it ?

Derivation Macro

We can use derivation macros with implicits

```scala
import sangria.macros.derive._

implicit val ReviewType: ObjectType[Unit, Review] = deriveObjectType[Unit, Review]()
implicit val ActorType: ObjectType[Unit, Actor] = deriveObjectType[Unit, Actor]()
implicit val TvShowType: ObjectType[Unit, TvShow] = deriveObjectType[Unit, TvShow]()

// Fonctionne aussi bien avec les Enumeration Scala que les sealed trait + case object
implicit val GenreEnum: EnumType[Genre.Value] = deriveEnumType[Genre.Value]()

implicit val MovieType: ObjectType[VideoContext, Movie] = deriveObjectType[VideoContext, Movie]()
```

Advantages : Less Code

Inconvenients: Less Flexibility

Derivation Object Type

```
implicit val MovieType: ObjectType[VideoContext, Movie] = deriveObjectType[VideoContext, Movie]()
```

```
trait VideoContext {
 val id: String
 val title: String
 val description: Option[String]
 val genres: Seq[Genre.Value]
 val actors: Seq[Actor]
 val rating: Option[Double]
 val reviews: Seq[Review]

 @GraphQLField
 def topActors(top: Int): Seq[Actor] =
actors.sortBy(_.popularity).reverse.take(top)

 @GraphQLField
 def lastReviews(last: Int): Seq[Review] =
reviews.takeRight(last)

}
```

```
case class Movie(id: String,
                 title: String,
                 description: Option[String],
                 genres: Seq[Genre.Value],
                 actors: Seq[Actor],
                 rating: Option[Double],
                 reviews: Seq[Review],
                 budget: Long,
                 gross: Seq[Long]
                ) extends VideoContext {

 @GraphQLField
 def profit(week: Int): Long = gross.take(week - 1).sum - budget

 @GraphQLField
 def alert: Boolean = gross.sum < 0

}
```

Derivation Object Type With Context

```scala
val QueryType = deriveContextObjectType[ApplicationContext, Query, Unit](_.QueryObject)
```

```scala
class ApplicationContext {

 object QueryObject extends Query

 object MutationObject extends Mutation

}
```

```scala
trait Query {

 @GraphQLField
 def movies: Vector[Movie] = MovieController.getAllMovies

 @GraphQLField
 def movie(id: String): Option[Movie] = MovieController.getMovieById(id)

 @GraphQLField
 def getMoviesByGenre(genres: Seq[Genre.Value]): Vector[Movie] =
MovieController.getMoviesByGenres(genres)

 @GraphQLField
 def movieByTitle(title: String): Option[Movie] = MovieController.getMovieByTitle(title)

}
```

Derivation Enum Type

Scala Enumeration

```scala
implicit val GenreEnum = deriveEnumType[Genre.Value]()


object Genre extends Enumeration {
 val ACTION = Value
 val COMEDY = Value
 val HORROR = Value
 val ANIMATION = Value
 val ADVENTURE = Value
 val SCI_FI = Value
}
```

Sealed trait + case object

```scala
implicit val GenreEnumSealed = deriveEnumType[GenreEnum]()


sealed trait GenreEnum

object GenreEnum {

 case object ACTION extends GenreEnum
 case object COMEDY extends GenreEnum
 case object HORROR extends GenreEnum
 case object ANIMATION extends GenreEnum
 case object ADVENTURE extends GenreEnum
 case object SCI_FI extends GenreEnum

}
```

Annotations

```
@GraphQLField: Specify a def as a Field

@GraphQLDescription: Describe a Field or a Type

@GraphQLDeprecated: Reason why the Field is deprecated

@GraphQLName: To rename a Field

@GraphQLExclude: Exclude a Field, Enum, Argument
```

Annotations

```scala
trait Query {

 @GraphQLField
 @GraphQLDescription("Fetch a specific movie by id")
 def movie(id: String): Option[Movie] = movies.find(_.id == id)

 @GraphQLField
 @GraphQLDescription("Fetch a specific by title")
 @GraphQLDeprecated("Please use movie field instead when possible")
 def movieByTitle(title: String): Option[Movie] = movies.find(_.title.toLowerCase() == title.toLowerCase)

}
```

How to interact with this schema ?

Interacting with GraphQL

How to interact with GraphQL ?

3 types of interactions:

- Query: Read-only
- Mutation: Create Update Delete
- Subscription: Real-Time update
  - Streams of data are sent to the subscribed client

A single endpoint */graphql*

Akka HTTP Server

POST /graphql => JSON

query, variables, operation that we parse with any JSON library

```
QueryParser.parse(query) match {
           case Success(queryAst) ⇒
            complete(Executor.execute(
                    schema = SchemaDefinitionMovie.GraphQLSchema,
                    queryAst = queryAst,
                    userContext = new ApplicationContext,
                    variables = vars,
                    operationName = operation
             ).map(OK → _))
           case Failure(error) ⇒
             complete(BadRequest, JsObject("error" → JsString(error.getMessage)))
         }
```

Application Context

```
// Root context of our application
class ApplicationContext {

 object QueryObject extends Query
 object MutationObject extends Mutation

}
```

GraphQL Schema

```
val QueryType = deriveContextObjectType[ApplicationContext, Query, Unit](_.QueryObject)
val MutationType = deriveContextObjectType[ApplicationContext, Mutation, Unit](_.MutationObject)

val GraphQLSchema = Schema(QueryType, Some(MutationType))
```

Type Query

```
val QueryType = deriveContextObjectType[ApplicationContext, Query, Unit](_.QueryObject)

@GraphQLDescription("This is our query type to fetch data")
trait Query {
 @GraphQLField
 @GraphQLDescription("Fetch all movies")
 def movies: Vector[Movie] = MovieController.getAllMovies

 @GraphQLField
 @GraphQLDescription("Fetch a specific movie by id")
 def movie(id: String): Option[Movie] = MovieController.getMovieById(id)

 @GraphQLField
 @GraphQLDescription("Fetch movies by genres")
 def getMoviesByGenre(genres: Seq[Genre.Value]): Vector[Movie] = MovieController.getMoviesByGenres(genres)

 @GraphQLField
 @GraphQLDescription("Fetch a specific by title")
 @GraphQLDeprecated("Please use movie field instead when possible")
 def movieByTitle(title: String): Option[Movie] = MovieController.getMovieByTitle(title)
}
```

Nested Fields

```scala
def movie(id: String): Option[Movie] = MovieController.getMovieById(id)

    def topActors(top: Int): Seq[Actor] = actors.sortBy(_.popularity).reverse.take(top)

        case class Actor(id: String,
                         fullname: String,
                         popularity: Option[Int]
                        )
```

```graphql
query {
  movie(id: "001") {
    topActors(top: 3) {
      id
      fullname
      popularity
    }
  }
}
```

Type Mutation

```scala
val MutationType = deriveContextObjectType[ApplicationContext, Mutation, Unit](_.MutationObject)


trait Mutation {

 @GraphQLField
 def createReview(movieId: String, author: String, rating: Double, content: String): Option[Movie] = {
   val review = Review(author = author, rating = rating, content = content)
   MovieController.addReview(movieId, review)
 }

}
```

Main difference between Query and Mutation

Because mutation modify a state, it is not possible to run them in parallel.

- Query Fields are run in parallel
- Mutation Fields are run in series

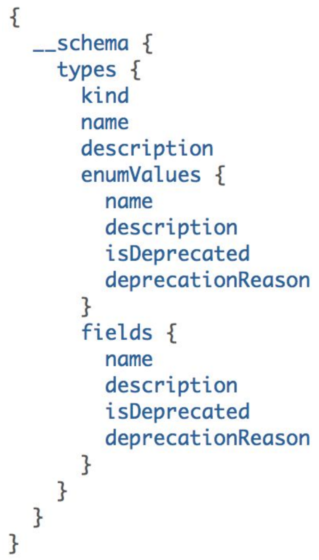
If we send two mutations in our query, they will be run one after the other

Client Side

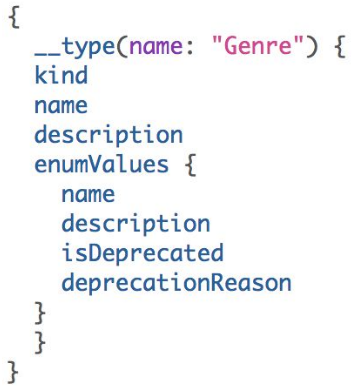
Introspection API

The Introspection API allows us to fetch graphql Schema in JSON

If we don’t know the schema, we can query it.



```
{
  __schema {
    types {
      kind
      name
      description
      enumValues {
        name
        description
        isDeprecated
        deprecationReason
      }
      fields {
        name
        description
        isDeprecated
        deprecationReason
      }
    }
  }
}
```



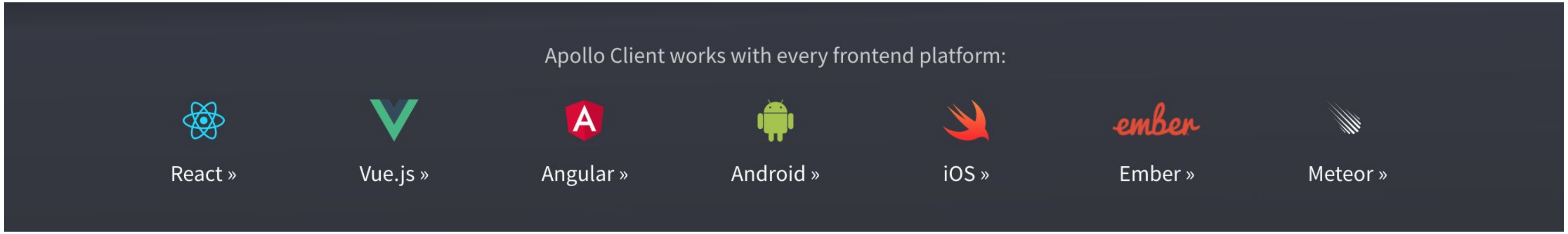
```
{
  __type(name: "Genre") {
  kind
  name
  description
  enumValues {
    name
    description
    isDeprecated
    deprecationReason
  }
  }
}
```

What kind of client consume GraphQL API

Most of the clients to a GraphQL server are **frontend applications**.

The GraphQL ecosystem client-side for frontend applications is extremely rich

For Example, Apollo Client which is really mature or Relay Client


Apollo Client works with every frontend platform:

What about Sangria client-side ?

Principal use cases are for tooling, validation and testing

- graphql macro can be used to validate query syntax at compile time

```
import sangria.macros._

test("query is valid") {

 val query =
   graphql"""
      query {
        movie(id: "001") {
          title
          description
          rating
          lastReviews(last: 5) {
            author
            content
          }
        }
      }
    """
 assert(QueryValidator.default.validateQuery(SchemaDefinitionMovie.GraphQLSchema, query) == Vector.empty[Violation])

}
```

What about Sangria client-side ?

- test graphql response for a query

```
test("query parsed correctly") {
 val query =
   graphql"""
      query {
        movie(id: "001") {
          title
        }
      }
    """
 assert(executeQuery(query).toString == """{"data":{"movie":{"title":"The Avengers"}}}""")
}
```

- retrieve graphql schema from sangria schema definition

```
SchemaRenderer.renderSchema(SchemaDefinitionMovie.GraphQLSchema)
```

Before We Go

When is GraphQL a good alternative to REST for your API ?

- Your API is used by mobile applications and data needs to be reached by the client even with a bad connection
- You have lots of different clients using your API and can’t customize your API for each one
- You have a backend team and a frontend team (they can work independently)

coursera SERVERLESS Product Hunt

yelp METEOR

Rest is still deeply rooted in our habits

But if you have the opportunity, why not use GraphQL

Resources

- www.howtographql.com (There is everything to know about GraphQL)
- http://facebook.github.io/graphql/ (Paper about GraphQL)
- http://sangria-graphql.org/learn/ (Sangria Documentation)
- https://github.com/sangria-graphql/sangria-akka-http-example (Boilerplate)