The first query

Goal

Our goal for this chapter is to run following query:

```
query {
  allLinks {
    id
    url
    description
  }
}
```

The expected response is a list of links fetched from database.

Define a model

before we will add the first model, please decide where all models will be stored. I have one recommendation but of course you can place all models in the place you want.

Create `models` package and `package.scala` with content:

```
package com.howtographql.scala.sangria

package object models {

}
```

From now on all the domain specific models will be placed in this file, so it will be easy to find them and in the case of this tutorial, easy to compare versions and recognize if something will need to change.

In the file created above add the following case class:

```
case class Link(id: Int, url: String, description: String)
```

As you can see, a `Link` model has fewer fields than in the schema you saw in the first chapter, but no worries, we will improve the model in the future. Now we're focusing on completing execution stack so it would be better to keep this model simple.

Add `Link` support to database

Our simple database has to support this model and provide some data as well.

Add following changes to the `DBSchema.scala` :

```
//in the imports section:
import com.howtographql.scala.sangria.models._

//In the object body:

//1
class LinksTable(tag: Tag) extends Table[Link](tag, "LINKS"){

    def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
    def url = column[String]("URL")
    def description = column[String]("DESCRIPTION")

    def * = (id, url, description).mapTo[Link]

}

//2
val Links = TableQuery[LinksTable]

//3
val databaseSetup = DBIO.seq(
    Links.schema.create,

    Links forceInsertAll Seq(
      Link(1, "http://howtographql.com", "Awesome community driven GraphQL tu
```

We just added database definition of our first model.

1 defines mapping to database table,

2 gives us a helper we will use to accessing data in this table.

The last change, **3** is responsible for creating schema and adding three entities to the database. Don't forget to replace the empty function which was provided by template

Providing GraphQL Context

Context is an object that flows across the whole execution, in the most cases this object doesn't change at all. The main responsibility of the Context is providing data and utils needed to fulfill the query. In our example we will put there `DAO` so all queries will have access to the database. In the future we will also put there authentication data.

In our example Context will get a name `MyContext` and because it isn't related with domain directly, I propose to keep it along other files in the `sangria` package.

Create `MyContext` class:

```
package com.howtographql.scala.sangria

case class MyContext(dao: DAO) {

}
```

For now we don't do anything with this file, but we had to create it to get the server working. For sure we will back to this file in the future.

GraphQL Server

execute it and through HTTP layer send response back to the client. It will also catch GraphQL parsing errors and convert those into the proper HTTP responses.

Create `GraphQLServer.scala` file:

```
package com.howtographql.scala.sangria

import akka.http.scaladsl.server.Route
import sangria.parser.QueryParser
import spray.json.{JsObject, JsString, JsValue}
import akka.http.scaladsl.model.StatusCodes._
import akka.http.scaladsl.server.Directives._
import scala.concurrent.ExecutionContext
import scala.util.{Failure, Success}
import akka.http.scaladsl.server._
import sangria.ast.Document
import sangria.execution._
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
import sangria.marshalling.sprayJson._


object GraphQLServer {

  // 1
  private val dao = DBSchema.createDatabase

  // 2
  def endpoint(requestJSON: JsValue)(implicit ec: ExecutionContext): Route =

    // 3
    val JsObject(fields) = requestJSON

    // 4
    val JsString(query) = fields("query")

    // 5
    QueryParser.parse(query) match {
      case Success(queryAst) =>
        // 6
        val operation = fields.get("operationName") collect {
          case JsString(op) => op
        }

        // 7
        val variables = fields.get("variables") match {
          case Some(obj: JsObject) => obj
          case _ => JsObject.empty
        }
        // 8
```

```
  }

  private def executeGraphQLQuery(query: Document, operation: Option[String],
    // 9
    Executor.execute(
      GraphQLSchema.SchemaDefinition, // 10
      query, // 11
      MyContext(dao), // 12
      variables = vars, // 13
      operationName = operation // 14
    ).map(OK -> _)
      .recover {
      case error: QueryAnalysisError => BadRequest -> error.resolveError
      case error: ErrorWithResolver => InternalServerError -> error.resolveEr
    }
  }

}
```

It's one of the most important files in entire backend server so let's analyze it step by step:

1 We need access to the database, so it's the step where such connection is created.

2 `endpoint` responds with `Route` type. It will be used directly in the routing of HTTP server. It expects JSON object as parameter.

3 Main JSON Object is extracted from the root object and it consists three children. The expected structure you can see in the following fragment

```
{
query: {},
variables: {},
operationName: ""
}
```

`query` is a query itself, `variables` is additional data for that query. In GraphQL you can send the query and arguments separately. You can also set name for the query, it's what the third object is for. Imagine that query is like a function, usually you're using anonymous functions, but for logging or other purposes you could add names. It's send as `operationName` .

`QueryParser.parse` (**5**) function we can use in this case. When it fails, the server will respond with status 400 and error description in the body of response. After successful parsing, we're also trying to extract the other two keys `operationName` (**6**) and `variables` (**7**). At the end all those three objects are passed to the execution function (**8**).

9 `Executor.execute` is the most important call in this class because it's the point where the query is executed. If the executor responds with success, the result is sent back to the client. In all other cases, the server will respond with status code 4xx and some kind of explanation of what was wrong with the query. The Executor needs some data to fulfill the request. Three of them are `query` (**11**), `operationName` (**13**) and `variables` (**14**) which are read from the request. The last two are: `GraphQLSchema.SchemaDefinition` and `MyContext(dao)` .

12 `MyContext` is a context object mentioned in the section above. In our example you can see that the context is built with the DAO object within.

`GraphQLSchema.SchemaDefinition` is the last object we have to explain here. It contains our Schema - what we are able to query for. It also interprets how data is fetched and which data source it could use (i.e. one or more databases, REST call to the other server...). In short our `SchemaDefinition` file defines what we want to expose. There are defined types (from GraphQL point of view) and shape of the schema a client is able to query for. Because this file is still missing we will create it in the next step.

Define GraphQLSchema

Create `GraphQLSchema` object:

```
package com.howtographql.scala.sangria

import sangria.schema.{Field, ListType, ObjectType}
import models._
// #
import sangria.schema._
import sangria.macros.derive._

object GraphQLSchema {

  // 1
  val LinkType = ObjectType[Unit, Link](
       "Link",
```

```
        )
      )

  // 2
  val QueryType = ObjectType(
    "Query",
    fields[MyContext, Unit](
      Field("allLinks", ListType(LinkType), resolve = c => c.ctx.dao.allLinks
    )
  )

  // 3
  val SchemaDefinition = Schema(QueryType)
}
```

Sangria cannot reuse case classes defined in our domain, it needs its own object of type `ObjectType`. On the other hand, it allows us to decouple API/Sangria models from database representation. This abstraction allows us to freely hide, add or aggregate fields.

1 is a definition of ObjectType for our `Link` class. First (String) argument defines the name in the schema. If you want it could differ from name of case class. In `fields` you have to define all those fields/functions you want to expose. Every field has to contain a `resolve` function which tells Sangria how to retrieve data for this field. As you can see there is also an explicitly defined type for that field. Manual mapping could be boring in cases where you have to map many case classes. To avoid boilerplate code you can use the provided macro.

```
implicit val LinkType = deriveObjectType[Unit, Link]()
```

It will give the same result as the example I used in the code above. When you want to use macro-way to define objects don't forget to import `sangria.macros.derive._`

2 `val QueryType` is a top level object of our schema. As you can see, the top level object has a name `Query` and it (along with nested objects) will be visible in the graphiql console that we will include later in this chapter. In `fields` definition I've added only one `Field` at the moment

The snippet above defines a GraphQL field with name `allLinks`. It's a list (ListType) of link items (LinkType). AS you can see it's a definition of a query we want to provide in this chapter. Resolver needs a `allLinks` function in `DAO` object so we have to implement it now.

Add a `allLinks` function to the `DAO` object, the current state of this file should looks like the following:

```
package com.howtographql.scala.sangria
import DBSchema._
import slick.jdbc.H2Profile.api._

class DAO(db: Database) {
  def allLinks = db.run(Links.result)
}
```

GraphiQL console

Graphiql makes able to run queries against our server from the browser. Let's implement it now.

Giter8 template I provided for this example also contains proper file. You can find it in `src/main/resources/graphiql.html`. All we need to do is to define the HTTP server is such way that this file will be exposed and available to be reached in the browser.

Configure HTTP Server endpoints

The last thing we have to do to fulfill this chapter's goal is to configure HTTP server. We have to expose `graphiql.html` file and open an endpoint where GraphQL queries will be send.

Open the `Server.scala` file and replace `route` function with the following one:

```
val route: Route =
    (post & path("graphql")) {
```

```
      getFromResource("graphiql.html")
    }
```

As you can see, a new `route` definition has only two endpoints. Every `POST` to `/graphql` is delegated to GraphQLServer, response to every other request else is a content of the `graphiql.html` file.

Run the query

Run the server

```
sbt run
```

And open in the browser an url http://localhost:8080/graphiql Of couse use different port number if you haven't decided to default one during project initialization.

In graphiql console execute the following code:

```
query {
  allLinks {
    id
    url
    description
  }
}
```

The response should looks like that:

```
{
  "data": {
```

```
        "description": "Awesome community driven GraphQL tutorial"
      },
      {
        "id": 2,
        "url": "http://graphql.org",
        "description": "Official GraphQL web page"
      },
      {
        "id": 3,
        "url": "https://facebook.github.io/graphql/",
        "description": "GraphQL specification"
      }
    ]
  }
}
```

Goal achieved

In this chapter we've finished configuring the entire GraphQL server stack and defined a very basic first query.

UNLOCK THE NEXT CHAPTER

Which key doesn't exists in the root JSON object.

- ~~operationName~~
- ~~variables~~
- fields
- query

Well done, you unlocked the next chapter!

In this chapter you will learn how to use arguments and how to handle them and pass to the business logic.

Go to next chapter

Edit on Github