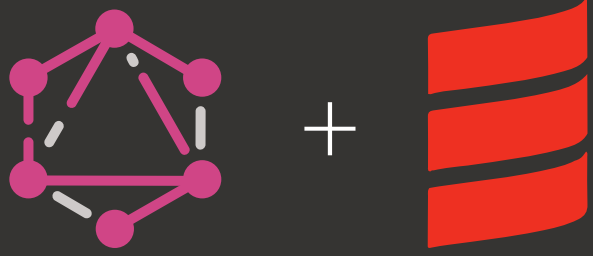
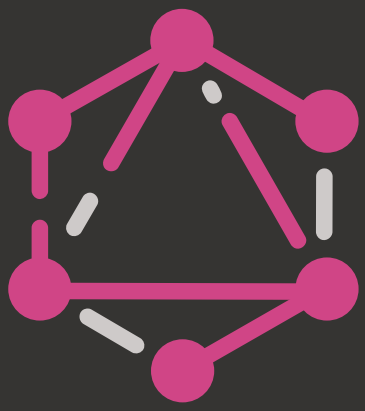


HOW TO GRAPHQL WITH SCALA



If you're interested
in learning GraphQL
for another languages visit:

howtographql.com

WRITTEN BY

Mariusz Nosiński

Fullstack Developer @ Scalac.io

Experienced Scala Developer who still enjoys learning new techs.

@marioosh

GraphQL Scala Tutorial:

What GraphQL is?

GraphQL is a specification developed by Facebook **which describes the new API syntax for managing backend data**. This syntax is in many ways a much more efficient and powerful alternative to REST. The main advantages of GraphQL are:

The client may specify exactly what data is needed.

GraphQL minimizes the amount of data transferred over a network noticeably improves application speed, it prevents underfetching. In REST some endpoints don't provide enough required data and the client has to make additional requests to get all needed information.

Aggregating data is easier.

You can develop a backend which is a facade for multiple data sources and with GraphQL, you can aggregate it transparently for the client. It's easier to get all related data in one query.

Data described in type system.

One of the advantages is that both teams (frontend and backend) could work independently if both sides know schema from the beginning. Adding new fields is almost transparent for the client, so you can evolve an API without changing any client's code! Specification also supports providing information about deprecation of fields - client is able to still use this field, but knows that it will be removed in the future and will be prepared for this change.

I would say that GraphQL Schema is a contract and API documentation in one file.

Some History and context

Before 2012 Facebook has put an effort into developing mobile client for their platform. Instead of sending a lot of requests,they have decided to go for a **Data-Driven API.**
Such API was supposed to be simple to learn and powerful to describe all of Facebook. GraphQL was born to fill all those needs. But Specification was patented in 2012 and many clients didn't use it because of that.

GraphQL became more and more popular over the years. In November 2017 Facebook decided to change the license to the MIT (https://opensource.org/licenses/MIT). Since then you can use it even for commercial purposes. Now, popularity of a specification and community around

graphql.org is growing continuously.

For example, You can find GraphQL as a common API standard for:

Github (https://developer.github.com/v4/)

Microsoft AzureWeb (https://graphql-demo.azurewebsites.net/)

Shopify (https://help.shopify.com/en/api/custom-storefronts/storefront-api/graphql-explorer)

Yelp (https://www.yelp.com/developers/graphiql)

(just naming a few above). To see more visit: **https://graphql.org/users/**

Basic comparison to REST

1. GraphQL and REST are totally different

2. Both aren't competitors, GraphQL is not a descendants.

3. You can use them simultaneously in one project.

4. Moreover, GraphQL isn't a magic bullet and...

5. One could be "better" than the other provided that they are used properly.

Core GraphQL Concept

GraphQL is a query string sent to the server. That query defines what data should be fetched and returned in JSON format.

GraphQL defines the shape of data. Response is mostly a mirror of a query, so you can predict what you'll get back from the server.

GraphQL has a hierarchical nature, it follows relationships between objects. Usually in REST you have to make multiple requests and then merge responses to achieve the same result.
Graph Databases uses similar structure internally and many of them provide

GraphQL-friendly interfaces for interchanging data.

GraphQL has defined a particular type on each level of query and each of that type has defined, available fields. It allows the client to verify a query before executing it. It also enables a query introspection and rich tooling e.g. autocompletion. Moreover, client could fetch this information (types/fields) beforehand, which is used by tools to speed up the developing process.

Finally user gets access to possible query structure and response types.

Introduction

Motivation

Scala is a very popular language nowadays and it's often chosen to deliver efficient and scalable systems. It leverages the Java VM, known for its reliability and robustness. Support for Functional Programming, rich ecosystem and stable foundation allow building fast applications, quickly.

In the next chapters you'll learn how to build your own GraphQL server using Scala and the following technologies:

> **Scala** — Scala language

> **Akka HTTP** — Web server to handle HTTP requests.

> **Sangria** — A library for GraphQL execution

> **Slick** — A Database query and access library.

> **H2 Database** — In-memory database.

> **Graphiql** — A simple GraphQL console to play with.

> **Giter8** — A project templating tool for Scala.

I assume you're familiar with GraphQL concepts, but if not, you can visit GraphQL site to learn more about that.

What is a GraphQL Server?

A GraphQL server should be able to:

> Receive requests following the GraphQL format, for example:

```
{  "query": "query { allLinks { url } }" }
```

> Connect to one or more data sources, like databases or other APIs and format obtained information.

> Response with the requested data, such as this one:

```
{
"data": { "allLinks": { "url": "http://graphql.org/" } }
}
```

> Validate incoming requests accordingly to the schema definition and supported formats.

For example, if a query is made with an unknown field, the response should be something like:

```
{
  "errors": [{
    "message": "Cannot query field \"unknown\" on type
\"Link\"."
  }]
}
```

As you can see our server will be really simple, but real GraphQL implementation can do much more than this. (We will explore it more later on.)

Schema-Driven Development

Schema-first GraphQL development forces frontend and backend developers to agree on a strict contract up front, enabling them to work quickly and efficiently while staying on spec. It improves both your API's performance and the performance of your team in general.

Sensibly then, the experience of building a GraphQL server starts with working on its schema.
You'll see in this chapter that the main steps you follow will be something like this:

1. Define your types and the appropriate queries and mutations for them.

2. Implement functions (called resolvers) to perform agreed upon queries in terms of defined types.

3. As new requirements arrive, go back to step 1 to update the schema and go through the other steps.

The schema is a contract between the frontend and backend, so keeping it at the center allows both sides of the development to evolve without going off the spec. This also makes it easier to parallelize the work. Since the frontend can move on

with a complete knowledge of the API from the start, using a simple mocking service (or even a full backend such as Graphcool) which can later be easily replaced with the final server.

Getting Started

In this chapter you will learn how to:

> Initialize the SBT project from *giter8* template.,

> Setup Database schema and connection.

Initialize new project

For the purpose of this tutorial I've prepared a *giter8* template. You can use it to easily bootstrap a project. All you need is the latest version of SBT.

Go to a directory where you want to bootstrap your project and please run this command:

```
sbt new marioosh/howtographql-scala-sangria.g8
```

You will be asked about project's name and port number to use by the HTTP server. You can hit ENTER to keep default values.

After this process you will see a simple project with the structure like this:

```
howtographql-sangria
├── README.md
├── build.sbt
├── project
│   ├── build.properties
│   └── plugins.sbt
└── src
    └── main
        ├── resources
        │   └── application.conf
        │   └── graphiql.html
        └── scala
            └── com
                └── howtographql
                    └── scala
                        └── sangria
                            ├── DAO.scala
                            ├── DBSchema.scala
                            └── Server.scala
```

I will explain shortly the most important files here.

> `build.sbt`

> `project/plugins.sbt`

> `project/build.properties`

Files above are related to SBT. There you can find all dependencies to external libraries and plugins we will be using in the project.

I assume you're at least a beginner in scala and you understand what is going on in those files. One thing you could be unfamiliar with is `Revolver` plugin.

This plugin is responsible for restarting server every time you save the files, so akka-http will always serve the updated version. It's very helpful during development.

HTTP Server

Open `Server.scala` file. It will be our HTTP server and entry point for the application.

Content of the file should look like this:

```
import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.server.Route
import akka.stream.ActorMaterializer
import akka.http.scaladsl.server.Directives._
import spray.json._
import
akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._

import scala.concurrent.Await
import scala.language.postfixOps
```

```
//1
object Server extends App {


  //2
  val PORT = 8080


  implicit val actorSystem = ActorSystem("graphql-server")
  implicit val materializer = ActorMaterializer()


  import actorSystem.dispatcher
  import scala.concurrent.duration._


  scala.sys.addShutdownHook(() -> shutdown())


  //3
  val route: Route = {

    complete("Hello GraphQL Scala!!!")

  }


  Http().bindAndHandle(route, "0.0.0.0", PORT)
  println(s"open a browser with URL: http://localhost:$PORT")

  def shutdown(): Unit = {

    actorSystem.terminate()
    Await.result(actorSystem.whenTerminated, 30 seconds)

  }
}
```

Our server extends an `App` trait so SBT can find it and run when you use `sbt run` command. All the `App` does is implement a `main` function which is a default entry point when it's executed.
In case there are more files like this in your project, SBT will ask you which one you want to run.

At the 2nd point, there is a defined port number we want to use, you could choose it during project initialization.

What is worth pointing out here: In our example I use Spray JSON library for marshalling and unmarshalling JSON objects, but it isn't

obligatory for you. You can use whichever JSON library you want. On this page you can find which JSON libraries Sangria can play with.

Database configuration

In our project I have chosen to use H2 database. It's easy to configure and is able to run in memory - you don't need to install any additional packages in your OS. H2 works perfectly in tutorials like this one. If you want to use another DB, it's up to you, Slick supports many of them.

```
h2mem = {
  url = "jdbc:h2:mem:howtographqldb"
  driver = org.h2.Driver
  connectionPool = disabled
  keepAliveConnection = true
}
```

It's all we need to configure the database. Now we're ready to use it. For the future purposes we will create two additional files.

`DAO.scala` is almost empty for now. It will be responsible for managing database connection.

In the second class: `DBSchema`, we will put a database schema configuration and some helper functions related to the data managment.

The object above will be useful in the future. We will use it to setup and configure the database.
For the sake of simplicity we won't worry too much about blocking.

To recap, in this chapter we have learnt how to:

> Initialize the SBT project,

> Setup Database schema and connection.

The first query

Goal

Our goal for this chapter is to run the following query:

```
query {
  allLinks {
    id
    name
    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 {

}
```

All of the domain's models will be placed in this file, so it will be easy to find them and in case of this tutorial easy to compare different versions and recognize if something changes.

Now we can get back to creating our first model.

Let's start by defining a really basic model `Link`.

In the file created above add the following case class:

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

As you can see, `Link` model model has less fields than in the schema you saw in the first chapter, but no worries, we will improve the model in the future. Now we will focus 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 tutorial"),

      Link(2, "http://graphql.org", "Official GraphQL web
page"),

      Link(3, "https://facebook.github.io/graphql/", "GraphQL
specification")

    )

)
```

We have just added the database definition of our first model.

> 1) defines mapping to database table,

> 2) gives us a helper we will use to access data in this table.

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

Define a model

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 to provide 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 to the domain directly, I suggest keeping it along with other files in the `sangria` package.

Create `MyContext` *class:*

```
package com.howtographql.scala.sangria

case class MyContext(dao: DAO) {

}
```

For now we do nothing with this file, but we had to create it to begin working with the server.
For sure we will return to this file in the future.

GraphQL Server

Time to implement the GraphQL Server. This object will be in the second layer of architecture, just after HTTP server. Proper HTTP request will be converted into JSON object and sent to this server. GraphQL Server will parse that JSON as GraphQL query, execute it and through HTTP layer send a response back to the client. It also catches GraphQL parsing errors and converts 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
        complete(executeGraphQLQuery(queryAst, operation,
variables))

      case Failure(error) =>

        complete(BadRequest, JsObject("error" ->
JsString(error.getMessage)))

    }

  }


  private def executeGraphQLQuery(query: Document, operation:
Option[String], vars: JsObject)(implicit ec: ExecutionContext)
= {

    // 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.resolveError

    }

  }

}
```

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

> 1 We need access to the database, so we create a connection.

> 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 of three children.

The expected structure can be seen 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 a 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 sent as `operationName`.

> 4 We're extracting `query` from a request at this point.

> When we have the query, we have to parse it. Sangria provides `QueryParser.parse` (5) function we can use in this case. When it fails, the server will respond with a "status 400" and an error description in the body of a 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 a success, the result is sent back to the client. In the other scenario 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...). To sum up `SchemaDefinition` file defines what we want to expose. There are defined types (from GraphQL point of view) and shapes 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",
       fields[Unit, Link](
         Field("id", IntType, resolve = _.value.id),
         Field("url", StringType, resolve = _.value.url),
         Field("description", StringType, resolve =
_.value.description)
       )
     )

  // 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`. 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 the ObjectType for our `Link` class. First (String) argument defines the name in the schema. It doesn't have to be the same as a case class name. In `fields` you have to define all 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 if you have to map many case classes. To avoid boilerplate code you can use the marco I have provided.

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

It will give the same result as the example have 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 the 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

```
Field("allLinks", ListType(LinkType), resolve = c =>
c.ctx.dao.allLinks)
```

The snippet above defines a GraphQL field with a 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 an `allLinks` function to the `DAO` object, the current state of this file should look 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 running queries against our server possible from the browser. Let's implement it now.

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

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

```
val route: Route =
    (post & path("graphql")) {
      entity(as[JsValue]) { requestJson =>
        GraphQLServer.endpoint(requestJson)
      }
    } ~ {
      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 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 a different port number if you haven't decided to use a default one during project initialization.

In graphiql console execute the following code:

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

The response should looks like that:

```
{

  "data": {
    "allLinks": [
      {
        "id": 1,
        "url": "http://howtographql.com",
        "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.

Handling Arguments

Arguments

Let's assume that we want to fetch the selected links using their ids.

Take this query for instance:

```
query {
    link(id: 1){
      id
      name
    }
    links(ids: [2, 3]){
      id
      name
    }
}
```

What must we do? Firstly add to `DAO` the functions that give us a link by one or more ID's.

Open the file `DAO.scala` and add the following functions:

```
def getLink(id: Int): Future[Option[Link]] = db.run(
  Links.filter(_.id === id).result.headOption
)

def getLinks(ids: Seq[Int]) = db.run(
  Links.filter(_.id inSet ids).result
)
```

Also don't forget to add the following imports:

```
import com.howtographql.scala.sangria.models.Link
import scala.concurrent.Future
```

Next, we have to add the fields to the main `Query` object and set the functions above as resolvers.

Now open the file `GraphQLSchema.scala` and add two additional definitions in the `fields` function of the `QueryType` object (just after `allLinks` field definition):

```
Field("link", //1
  OptionType(LinkType), //2
  arguments = List(Argument("id", IntType)), //3
  resolve = c => c.ctx.dao.getLink(c.arg[Int]("id")) //4
),

Field("links", //1
  ListType(LinkType), //2
  arguments = List(Argument("ids", ListInputType(IntType))), //3
  resolve = c => c.ctx.dao.getLinks(c.arg[Seq[Int]]("ids")) //4

)
```

Let's try to understand what is going on:

- As explained previously, we add new fields with these names (`link` and `links`)

- The second parameter is the expected output type. In the first query it's an Optional Link, in second it's a list of links.

- `arguments` is a list of expected arguments defined by a name and type. In the first field, we're expecting an id argument of `Int` type. In the second case, we're expecting `ids` as a list of integers. As you can see we didn't use `ListType` in that case. We've used `ListInputType` instead. The main difference is that all `InputTypes` are used to parse incoming data, and `ObjectTypes` (mostly) are used for outgoing data.

- `arguments` defines which arguments we expect. Mostly, it is not forgotten and should be extracted and passed down to the resolver. `Context` object, reachable in `resolve` partial function, contains such information, so you have to fetch those arguments from there.

DRY with arguments

The code above can be a little simplified. You can extract an Argument as a constant and reuse this in the field declaration. You can change the link declaration like this:

```
val Id = Argument("id", IntType)

Field("link",
      OptionType(LinkType),
      arguments = Id :: Nil, // it's a list!
      resolve = c => c.ctx.dao.getLink(c.arg(Id))
)
```

You can make a similar change to the `links` field too. After these changes `GraphQlSchema` file should looks like this:

```
package com.howtographql.scala.sangria

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

object GraphQLSchema {

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

  val Id = Argument("id", IntType)
  val Ids = Argument("ids", ListInputType(IntType))

  val QueryType = ObjectType(
    "Query",
    fields[MyContext, Unit](
      Field("allLinks", ListType(LinkType), resolve = c =>
c.ctx.dao.allLinks),
      Field("link",
        OptionType(LinkType),
        arguments = Id :: Nil,
        resolve = c => c.ctx.dao.getLink(c.arg(Id))
      ),
      Field("links",
        ListType(LinkType),
        arguments = Ids :: Nil,
        resolve = c => c.ctx.dao.getLinks(c.arg(Ids))
      )))

  val SchemaDefinition = Schema(QueryType)
}
```

Now, we have a few fields. We're able to fetch either a single link or a list of chosen links.

Open the graphiql console in the browser and execute the following query:

```
query {
    link(id: 1){
      id
      url
    }
    links(ids: [2,3]){
      id
      url
    }
}
```

As a result you should see a proper output what was the goal for the chapter.

BTW, what about such query?

```
query {
  l1: link(id: 1){
    id
    url
  }
  l2: link(id: 1){
    id
    url
  }
}
```

If you debug the DAO class you will find out that `getLink` is called twice for the same `id`. `resolve` function and calls that function directly, so it's being called upon every `id`. Better way to execute that. Sangria provides a mechanism which helps to optimize or cache queries. This is exactly what we need here. So, after defining a problem you can switch to the next chapter and learn how to fix it.

Deferred Resolvers

Deferred Resolvers and Fetchers

`Fetchers` and `Deferred Resolvers` are mechanisms for batch retrieval of objects from their sources like database or external API. `Deferred Resolver` provides an efficient, low-level API, but it's more complicated to use and less secure. `Fetcher` is a specialized version of `Deferred Resolver`. It provides a high-level API, it's easier to use and usually provides all you need. It optimizes resolution of fetched entities based on ID or relation. Also, it deduplicates entities and caches the results and much more.

Let's implement one for the `Link` entity:

In `GraphQLSchema` file add a fetcher for `Link` entity. Place it before `QueryType` declaration and don't forget about imports:

```
//in import section:
import sangria.execution.deferred.Fetcher

//in the body:
val linksFetcher = Fetcher(
  (ctx: MyContext, ids: Seq[Int]) => ctx.dao.getLinks(ids)
)
```

Now we have to change fields in `QueryType` to use that Fetcher:

Change fields definition for `link` and `links` with the following code:

```
Field("link",
  OptionType(LinkType),
  arguments = Id :: Nil,
  resolve = c => linksFetcher.defer(Id)
),
Field("links",
  ListType(LinkType),
  arguments = List(Argument("ids", ListInputType(IntType))),
  resolve = c => linksFetcher.deferSeq(Ids)
)
```

We're still using `dao.getLinks` to fetch links from database, but now it's wrapped in `Fetcher`. It optimizes the query **before** call. Firstly it gathers all data it should fetch and then it executes queries. Caching and deduplication mechanisms allow to avoid duplicated queries and give results faster.

As you can see, we use the same fetcher in two fields, in the first example we only provide a single id and expecting one entity in response, optional object

(`deferOpt` function). In the second case we're providing a list of ids and expecting a sequence of objects (`deferSeq`).

After defining a resolver we have to inform an executor about it. Firstly, push it to the lower level by using proper `DeferredResolver` function which rebuilds it:

In `GraphQLSchema` create a constant for the deferred resolver.

```
//add to imports:
import sangria.execution.deferred.DeferredResolver


//add after linksFetcher:
val Resolver = DeferredResolver.fetchers(linksFetcher)
```

Such resolver has to be passed into the `Executor` to make available to use.

Add the resolver to the executor, open the `GraphQLServer.scala` file, and change `executeGraphQLQuery` function content as follows:

```
Executor.execute(
  GraphQLSchema.SchemaDefinition,
  query,
  MyContext(dao),
  variables = vars,
  operationName = operation,
  deferredResolver = GraphQLSchema.Resolver
)
//the rest without changes
```

Since, we're using `DAO.getLinks` to fetch a single entity or an entire list, we don't need the `getLink` function anymore.

Open a `DAO` class and remove the `getLink` function

HasId type class

If you tried to execute a query, you would get an error at this point. The reason is that `Fetcher` needs 'something' that extracts ids from entities. This thing is `HasId` type class. You have a few choices on how to provide such class for your model. Firstly you can pass it explicitly, like this:

```
val linksFetcher = Fetcher(
  (ctx: MyContext, ids: Seq[Int]) => ctx.dao.getLinks(ids)
)(HasId(_.id))
```

Another way is to declare an implicit constant in the same context so the fetcher will take it implicitly. The third way is to provide `HasId` implicitly inside the companion object of our model, like this:

```
object Link {
    implicit val hasId = HasId[Link, Int](_.id)
}
```

For now we're going down the second path, but in the `Interfaces` chapter it will change.

Add an implicit hasId in `GraphQLSchema`

```
//add to imports section:
import sangria.execution.deferred.HasId

//add just below LinkType declaration
implicit val linkHasId = HasId[Link, Int](_.id)
```

After the last changes, our `GraphQLSchema` file content should look like the this:

```
package com.howtographql.scala.sangria

import sangria.schema.{ListType, ObjectType}
import models._
import sangria.execution.deferred.{DeferredResolver, Fetcher,
HasId}
import sangria.schema._
import sangria.macros.derive._

object GraphQLSchema {

  implicit val LinkType = deriveObjectType[Unit, Link]()
  implicit val linkHasId = HasId[Link, Int](_.id)

  val linksFetcher = Fetcher(
    (ctx: MyContext, ids: Seq[Int]) => ctx.dao.getLinks(ids)
  )

  val Resolver = DeferredResolver.fetchers(linksFetcher)

  val Id = Argument("id", IntType)
  val Ids = Argument("ids", ListInputType(IntType))
```

```
  val QueryType = ObjectType(
    "Query",
    fields[MyContext, Unit](
      Field("allLinks", ListType(LinkType), resolve = c =>
c.ctx.dao.allLinks),
      Field("link",
        OptionType(LinkType),
        arguments = Id :: Nil,
        resolve = c => linksFetcher.deferOpt(c.arg(Id))
      ),
      Field("links",
        ListType(LinkType),
        arguments = Ids :: Nil,
        resolve = c => linksFetcher.deferSeq(c.arg(Ids))
      )
    )
  )

  val SchemaDefinition = Schema(QueryType)

}
```

Test it

You can debug the `DAO.getLinks` function in any way and execute the following query:

```
query {

  l1: link(id: 1){
      id
      url
    }

  l2: link(id: 1){
      id
      url
    }

  links(ids: [2,3]){
      id
      url
  }
}
```

Even though the query requests the same link twice, when you debug it, you can see that `getLinks` is called only once for that ID!

Goal for this chapter

To match the schema from the first chapter we have to extend a `Link` model with an additional field: `createdAt`. This field will store information about date and time. The problem is that `H2` database understands only timestamp. Sangria has a similar limitation - it supports only basic types. Our goal is to store the date and time information in the database and to present it in a human friendly format.

Extend a Link model

The type for storing date and time I chose is `akka.http.scaladsl.model.DateTime`. It fits our example because it has implemented ISO Format converters. (This type covers all my needs without any additional work. So I chose it, but in real production application avoid this if you can. Java has dedicated package for date and time and it includes many classes you can use.)

Change the content of `Link.scala`*:*

```
//in imports:
import akka.http.scaladsl.model.DateTime

//replace current case class
case class Link(id: Int, url: String, description: String,
createdAt: DateTime)
```

Fix the database

Currently in `DbSchema` we're storing few links in database. We have to add the additional field.

Add the `createdAt` *field in the* `Link` *models for populated example data in* `DBSchema`*. Change the* `databaseSetup` *function into the following code:*

```
Links forceInsertAll Seq(
      Link(1, "http://howtographql.com", "Awesome community
driven GraphQL tutorial", DateTime(2017,9,12)),


      Link(2, "http://graphql.org", "Official GraphQL web
page",      DateTime(2017,10,1)),


      Link(3, "https://facebook.github.io/graphql/", "GraphQL
specification", DateTime(2017,10,2))
    )
```

Almost good, but H2 doesn't know how to store such type in database, so we will instruct it how to store it using built-in types.

Add column mapper for our DateTime type in DBSchema object (before the LinksTable definition).

```
//ads to imports:
import java.sql.Timestamp

//and at the begining of the class' body:
implicit val dateTimeColumnType =
MappedColumnType.base[DateTime, Timestamp](
    dt => new Timestamp(dt.clicks),
    ts => DateTime(ts.getTime)
)
```

This mapper will convert DateTime into Long, which is a primitive recognized by H2.

The last thing is to add the createdAt column definition in the table declaration.

*Add the following code inside LinksTable class. Replace the current * function with the following one.*

```
def createdAt = column[DateTime]("CREATED_AT")

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

In current state the DBSchema file should have following content:

```
package com.howtographql.scala.sangria

import java.sql.Timestamp
import akka.http.scaladsl.model.DateTime
import com.howtographql.scala.sangria.models._
import slick.jdbc.H2Profile.api._
import scala.concurrent.duration._
import scala.concurrent.Await
import scala.language.postfixOps

object DBSchema {

  implicit val dateTimeColumnType =
MappedColumnType.base[DateTime, Timestamp](
    dt => new Timestamp(dt.clicks),
    ts => DateTime(ts.getTime)
  )
```

```
  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 createdAt = column[DateTime]("CREATED_AT")

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

  }


  val Links = TableQuery[LinksTable]

  /**
    * Load schema and populate sample data within this
Sequence of DBActions
    */

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

    Links forceInsertAll Seq(

      Link(1, "http://howtographql.com", "Awesome community
driven GraphQL tutorial", DateTime(2017,9,12)),

      Link(2, "http://graphql.org", "Official GraphQL web
page", DateTime(2017,10,1)),

      Link(3, "https://facebook.github.io/graphql/", "GraphQL
specification", DateTime(2017,10,2))

    )
  )

  def createDatabase: DAO = {

    val db = Database.forConfig("h2mem")

    Await.result(db.run(databaseSetup), 10 seconds)
    new DAO(db)
  }

}
```

Define custom scalar for DateTime

Sangria supports scalars for basic GraphQL types like `String`, Int, etc. In addition you can find scalars for types like `Long`, `BigInt` or `BigDecimal`. There are a lot of them, but you might encounter situations where custom or unsupported types should be used. Like in our example, we use `DateTime` type and there are no built-in scalars for those. To add support for our case, we will use the same trick as with H2. We will define conversions between type we want and type Sangria understand and then back again to our type. The best underlying type would be String, so we will use it in our implementation.

Let's write a scalar that converts `String` to `DateTime` and vice versa.

In `GraphQLSchema` add the following code:

```
//imports:
import sangria.ast.StringValue

//begining of the object's body:
implicit val GraphQLDateTime = ScalarType[DateTime](//1
  "DateTime",//2
  coerceOutput = (dt, _) => dt.toString, //3
  coerceInput = { //4

    case StringValue(dt, _, _ ) =>
DateTime.fromIsoDateTimeString(dt)
   .toRight(DateTimeCoerceViolation)

    case _ => Left(DateTimeCoerceViolation)
  },

  coerceUserInput = { //5

    case s: String => DateTime.fromIsoDateTimeString(s)
      .toRight(DateTimeCoerceViolation)

    case _ => Left(DateTimeCoerceViolation)
  }
)
```

> 1) Use `implicit` because it implicitly has to be in scope

> 2) `"DateTime"`. The name will be used in schemas.

> 3) `coerceOutput` converts our type to a String. It will be used to produce the output data.

> 4) `coerceInput` needs a partial function with `Value` as single argument. Such value could be of many types. In our case we're parsing only from `StringValue`. Of course nothing stops you from defining many

conversions. If you define more cases for `coerceInput`, users will have the freedom to provide input in more ways.

> 5) `coerceUserInput` converts literal which almost always is a String. While this function should cover basic types, `coerceInput` and `coerceOutput` should always be a value that the GraphQL grammar supports.

Both functions `coerceInput` and `coerceUserInput` should respond with `Either`. The correct (right) value should consist an object of expected type. In case of failure, the left value should contain a `Violation` subtype. Sangria provides many `Violation` subtypes, but in the code above you can see I've used `DateTimeCoerceViolation`. Let's implement this.

Add in the models package object the following definition:

```
//imports header:
import sangria.validation.Violation

//package body:
  case object DateTimeCoerceViolation extends Violation {

    override def errorMessage: String = "Error during parsing
DateTime"

  }
```

Finally, add the new field definition to the `LinkType.`

Replace `LinkType` definition in the `GraphQLSchema` file with the following piece of code:

```
  val LinkType = ObjectType[Unit, Link](
      "Link",
      fields[Unit, Link](
        Field("id", IntType, resolve = _.value.id),
        Field("url", StringType, resolve = _.value.url),
        Field("description", StringType, resolve =
_.value.description),
        Field("createdAt", GraphQLDateTime, resolve=
_.value.createdAt)
      )
    )
```

In case you've used derived version of `LinkType` you have to manage this additional field another way. Because Macro doesn't know the type you want to use you have to define it explicitly. Good news is that you have to define only this

field - the rest is still managed by macro.

In case of derived LinkType, the definition should looks like this:

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

    ReplaceField("createdAt", Field("createdAt",
GraphQLDateTime, resolve = _.value.createdAt))

  )
```

After the last changes `GraphQLSchema` file should looks like this:

```
package com.howtographql.scala.sangria

import akka.http.scaladsl.model.DateTime
import sangria.schema.{ListType, ObjectType}
import models._
import sangria.ast.StringValue
import sangria.execution.deferred.{DeferredResolver, Fetcher,
HasId}
import sangria.schema._
import sangria.macros.derive._

object GraphQLSchema {

  implicit val GraphQLDateTime = ScalarType[DateTime](//1
    "DateTime",//2
    coerceOutput = (dt, _) => dt.toString, //3
    coerceInput = { //4

      case StringValue(dt, _, _ ) =>
DateTime.fromIsoDateTimeString(dt)
        .toRight(DateTimeCoerceViolation)

      case _ => Left(DateTimeCoerceViolation)

    },

    coerceUserInput = { //5

      case s: String => DateTime.fromIsoDateTimeString(s)
                        .toRight (DateTimeCoerceViolation)

      case _ => Left(DateTimeCoerceViolation)

    }
  )
```

```
  val LinkType = deriveObjectType[Unit, Link](
    ReplaceField("createdAt", Field("createdAt",
GraphQLDateTime, resolve = _.value.createdAt))
  )


  implicit val linkHasId = HasId[Link, Int](_.id)

  val linksFetcher = Fetcher(
    (ctx: MyContext, ids: Seq[Int]) => ctx.dao.getLinks(ids)
  )

  val Resolver = DeferredResolver.fetchers(linksFetcher)

  val Id = Argument("id", IntType)
  val Ids = Argument("ids", ListInputType(IntType))


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

      Field("link",
        OptionType(LinkType),
        arguments = Id :: Nil,
        resolve = c => linksFetcher.deferOpt(c.arg(Id))
      ),

      Field("links",
        ListType(LinkType),
        arguments = Ids :: Nil,
        resolve = c => linksFetcher.deferSeq(c.arg(Ids))
      )
    )
  )

  val SchemaDefinition = Schema(QueryType)

}
```

Now when you'll add `createdAt` field to the query, you should get proper response. For example on query:

```
query {

  link(id: 1){
    id
    url
    createdAt
    }
  }
```

You will get a response:

```
{
  "data": {
    "link": {
      "id": 1,
      "url": "http://howtographql.com",
      "createdAt": "2017-09-12T00:00:00"
    }
  }
}
```

We've achieved our goal!!

Now you know the basics. In the next chapter you will add additional models. We will extract some common parts as interface to keep them in one place.

Interfaces

At this point you should know the basics, so it's the perfect time for some hands-on training. The following paragraph will give you hints on what needs to be done. Try implementing it yourself. At the end I will add an example solution in case you are stuck. Later in this chapter we will learn about interfaces and how they relate to the work you've done.

Your DIY kit

Before you go further, try to implement the changes yourself. I think, at this point, you have the necessary knowledge to add the `User` and `Vote` models. I'll show what to do later in this chapter, but try to implement it yourself first.

What you have to do:

– Add `User` class with fields: `id, name, email, password` and `createdAt`

– Add `Vote` class with fields: `id, createdAt, userId, linkId`(you don't have to to define any relations for now)

–Create database tables for both,

– Add object types for both,

– Add fetchers for both,

– Implement `HasId` type class,

– Add fields in main `ObjectType` which allows for fetching a list of entities like `users` and `votes`

Please, go ahead with your implementation... I will wait here

User entity

Let's start from the user entity:

Add `User.scala`*class to* `models` *package object with the following content:*

```
case class User(id: Int, name: String, email: String,
password: String, createdAt: DateTime = DateTime.now)
```

Database setup.

Add the following content to the `DBSchema` *class (after* `Links` *definition):*

```
class UsersTable(tag: Tag) extends Table[User](tag, "USERS"){

  def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
  def name = column[String]("NAME")
  def email = column[String]("EMAIL")
  def password = column[String]("PASSWORD")
  def createdAt = column[DateTime]("CREATED_AT")

  def * = (id, name, email, password, createdAt).mapTo[User]
}

val Users = TableQuery[UsersTable]
```

Sample entities:

In DBSchema in the function `databaseSetup`*, add an action* `Users.schema.create` *at beginning of the function and then add a few users later in this function:*

```
Users forceInsertAll Seq(

    User(1, "mario", "mario@example.com", "s3cr3t"),
    User(2, "Fred", "fred@flinstones.com", "wilmalove")

  )
```

Add a function responsible for user retrieval:

In `DAO` *class add a function:*

```
def getUsers(ids: Seq[Int]): Future[Seq[User]] = {

    db.run(
      Users.filter(_.id inSet ids).result
    )
}
```

Dont' forget about `import com.howtographql.scala.sangria.models.User`...

GraphQL part:

In `GraphQLSchema` add :

```
//ObjectType for user
val UserType = deriveObjectType[Unit, User]()

//HasId type class
implicit val userHasId = HasId[User, Int](_.id)

// resolver
val usersFetcher = Fetcher(
    (ctx: MyContext, ids: Seq[Int]) => ctx.dao.getUsers(ids)
)
```

Add fetcher to resolvers.

And lastly add the new fetcher to the resolvers list. In the same file, replace constant `Resolver` with:

```
val Resolver = DeferredResolver
                        .fetchers(linksFetcher, usersFetcher)
```

Add fields to main ObjectType:

Add to `QueryType.fields`:

```
Field("users",
        ListType(UserType),
        arguments = List(Ids),
        resolve = c => usersFetcher.deferSeq(c.arg(Ids))
)
```

We're ready... you can now execute a query like this:

```
query {
    users(ids: [1, 2]){
      id
      name
      email
      createdAt
    }
}
```

Vote entity

If you want, you can make similar changes for `Vote` entity. And then follow the instructions and check wheter everything works.

Create Vote class:

```
case class Vote(id: Int, userId: Int, linkId: Int, createdAt:
DateTime = DateTime.now)
```

Database setup.

Add the following content to the `DBSchema` class:

```
class VotesTable(tag: Tag) extends Table[Vote](tag, "VOTES"){

    def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
    def userId = column[Int]("USER_ID")
    def linkId = column[Int]("LINK_ID")
    def createdAt = column[DateTime]("CREATED_AT")

    def * = (id, userId, linkId, createdAt).mapTo[Vote]
  }

  val Votes = TableQuery[VotesTable]
```

The next step is creating relations.

In file `DBSchema` in function `databaseSetup`: Add an action `Votes.schema.create` at beginning of the sentence and then add few users later in this function:

```
Votes forceInsertAll Seq(
  Vote(id = 1, userId = 1, linkId = 1),
  Vote(id = 2, userId = 1, linkId = 2),
  Vote(id = 3, userId = 1, linkId = 3),
  Vote(id = 4, userId = 2, linkId = 2),
)
```

Add votes retrieval function.

In `DAO` class add a function:

```
def getVotes(ids: Seq[Int]): Future[Seq[Vote]] = {
    db.run(
      Votes.filter(_.id inSet ids).result
    )
}
```

GraphQL part:

In `GraphQLSchema` add :

```
implicit val VoteType = deriveObjectType[Unit, Vote]()
implicit val voteHasId = HasId[Vote, Int](_.id)

val votesFetcher = Fetcher(
  (ctx: MyContext, ids: Seq[Int]) => ctx.dao.getVotes(ids)
)
```

Add fetcher to resolvers.

Add the new fetcher to the resolvers list. In the same file, replace constant `Resolver` with:

```
val Resolver = DeferredResolver.fetchers(linksFetcher,
usersFetcher, votesFetcher)
```

Add fields to main ObjectType:

Add to `QueryType.fields`:

```
Field("votes",
        ListType(VoteType),
        arguments = List(Ids),
        resolve = c => votesFetcher.deferSeq(c.arg(Ids))
)
```

Don forget about `Ids` argument:

```
val Ids = Argument("ids", ListInputType(IntType))
```

The following query should now execute successfully:

```
query {
  votes(ids: [1, 2]){
    id
    createdAt
  }
}
```

Finding common parts

As you can see some code is very similar. Like `HasId` for all three types:

```
implicit val linkHasId = HasId[Link, Int](_.id)
implicit val userHasId = HasId[User, Int](_.id)
implicit val voteHasId = HasId[Vote, Int](_.id)
```

What if you want to add more entities? You will have to duplicate this code.

The solution for this is an interface. We can provide an interface that will be extended by any of the entities. This way, for example, you will need just one `HasId`

Create trait `Identifiable` int he `models` package object:

```
trait Identifiable {
  val id: Int
}
```

And then extend this trait by all of those classes like:

```
case class Link(...) extends Identifiable
case class User(...) extends Identifiable
case class Vote(...) extends Identifiable
```

Now we can replace all of the above `HasId` type classes with a single one. But now we will move it into companion object so it will be accessible whenever we import the trait.

Remove `linkHasId`, `userHasId` and `voteHasId`, and add companion object to the `Identifiable` trait:

```
//add to imports:
import sangria.execution.deferred.HasId

//add int he body
object Identifiable {
    implicit def hasId[T <: Identifiable]: HasId[T, Int] =
HasId(_.id)
}
```

The next step is providing GraphQL's interface type for that trait.

Add a definition of the interface and change the `LinkType` *for the following:*

```
val IdentifiableType = InterfaceType(
  "Identifiable",
  fields[Unit, Identifiable](
    Field("id", IntType, resolve = _.value.id)
  )
)

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

Make similar changes to `UserType` and `VoteType`.

Now if you look into the schema definition in graphiql console you will see that all three models implement the `Identifiable` interface.

So far so good. We made many changes in this chapter, so if you like you can compare current state to files with the following snippets.

Relations

Relations define how entities are connected with one another. You probably encountered those while working with databases. In GraphQL (and Sangria) relations are strictly connected with deferred resolvers and have a similar role. When you want to find related entities, the query can be optimized and all needed data fetched at once.

In other words: Relations expand Fetchers, allows for finding entities not only by their id field, by also by ids quite often stored in fields of another entity.

Lets try to define how many relations we have in our schema.

`User` has `links` and `votes` fields.

`Link` has `postedBy` and `votes` fields.

`Vote` has `user` and `link`

How do those relations work? `Link` is a main entity, first created by us and the most important. `Link` is added by a (single) user. On the other hand, a user can have more than one link.

`User` also can vote for a link. He can vote for a single link once, but a link can have more than one votes.

So, in our app we have 3 one-to-many relations.

Preparing database

First change slightly `Link` model:

Add `postedBy` field to the `Link` case class, the class should look like this:

```
case class Link(id: Int, url: String, description: String,
postedBy: Int, createdAt: DateTime = DateTime.now) extends
Identifiable
```

Update `LinksTable`.

Change the database schema. In the `DBSchema` change `LinksTable`, apply changes from the following code:

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

    // ...
    def postedBy = column[Int]("USER_ID")

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

}
```

Add foreign keys.

In `LinksTable` *add:*

```
def postedByFK =
         foreignKey("postedBy_FK", postedBy, Users)(_.id)
```

`Votes` model already has proper fields for storing external ids, we only have to add foreign keys in database setup.

In `VotesTable` *add:*

```
 def userFK = foreignKey("user_FK", userId, Users)(_.id)
 def linkFK = foreignKey("link_FK", linkId, Links)(_.id)
```

Because domain models have slightly changed, we also have to redefine our data.

`databaseSetup` *should be changed as in the following code:*

```
/**
  * Load schema and populate sample data within this Sequence
of DBActions
  */

val databaseSetup = DBIO.seq(

  Users.schema.create,
  Links.schema.create,
  Votes.schema.create,

  Users forceInsertAll Seq(
    User(1, "mario", "mario@example.com", "s3cr3t"),
    User(2, "Fred", "fred@flinstones.com", "wilmalove")
  ),
```

```
  Links forceInsertAll Seq(

    Link(1, "http://howtographql.com", "Awesome community
driven GraphQL tutorial",1, DateTime(2017,9,12)),
    Link(2, "http://graphql.org", "Official GraphQL web
page",1, DateTime(2017,10,1)),
    Link(3, "https://facebook.github.io/graphql/", "GraphQL
specification",2, DateTime(2017,10,2))
  ),

  Votes forceInsertAll Seq(
    Vote(1, 1, 1),
    Vote(2, 1, 2),
    Vote(3, 1, 3),
    Vote(4, 2, 2),
  )
)
```

I think we're done with Database part of changes.

No we can go and do a GraphQL part of changes.

Defining User->Link relation

Lets begin with User-Link relation. In the first entity we have to add the field `links` and in the second the field `postedBy`. Both fields uses the same Relation model.

Actually a `Link` entity has to have two defined relations. Firstly, because we can lookup database to find a link with particular Id, secondly, when we want to filter links by user ids stored in `postedBy` column. Our Fetcher accepts the provided id already what covers the first case but we still have to define the second one:

In `GraphQLSchema` Add a relation to be able to find `Links` by userId.

```
//add to imports:
import sangria.execution.deferred.Relation

//place before fetchers definition
val linkByUserRel =
      Relation[Link, Int]("byUser", l => Seq(l.postedBy))
```

This relation is of type `SimpleRelation` and has only two arguments: the first is the name, the second is a function which extracts a sequence of user ids from the link entity. Our case is super easy, because `postedBy` has such an id. All we need to do is wrap it into a sequence.

Now we have to add this relation to the fetcher. To do this, we have to use `Fetcher.rel` function instead of the previously used `apply`

Change `linksFetcher` to the following:

```
//add to imports:
import sangria.execution.deferred.RelationIds

//replace the current `linksFetcher` declaration
val linksFetcher = Fetcher.rel(

    (ctx: MyContext, ids: Seq[Int]) => ctx.dao.getLinks(ids),
    (ctx: MyContext, ids: RelationIds[Link]) =>
ctx.dao.getLinksByUserIds(ids(linkByUserRel))
  )
```

What do we have here? As I mentioned above, now we're using `.rel` function. It needs the second function to be passed as the argument. This function is for fetching related data from a datasource. In our case it uses a function `getLinksByUserIds` that we have to add to our dao. `ids(linkByUserRel)` extracts user ids by the defined in relation way and passes it into the DAO function.

In `DAO` class add a function:

```
def getVotes(ids: Seq[Int]): Future[Seq[Vote]] = {

    db.run(
      Votes.filter(_.id inSet ids).result
    )
}
```

Actually we've simplified the code above a little. When you look into the part `ctx.dao.getLinksByUserIds(ids(linkByUserRel))` a bit, you can wonder "And what if link has two relations? Could `getLinkByUserIds` be replaced by another function?" Be patient, such case will be covered later in this chapter. In our case we have only one relation, so we can retrieve all `userId`'s by calling `ids(linkByUserRel)` functions.

Add fields to GraphQL Objects

Let's begin with `LinkType`. `Link` already has a `postedBy` field, but for now it's only an `Int` and we need the entire user. To achieve this we have to replace the entire field definition and instruct resolver to use already defined fetcher to do this.

Add `ReplaceField` type class to the `LinkType` constructor.

```
ReplaceField("postedBy",
      Field("postedBy", UserType, resolve = c =>
usersFetcher.defer(c.value.postedBy))
)
```

In similar way we will change the `UserType` but `User` entity doesn't have `links` property so we have to add such field manually to the ObjectType. `AddField` type class is for such reason:

```
AddFields(
  Field("links", ListType(LinkType),
  resolve =
    c =>  linksFetcher.deferRelSeq(linkByUserRel, c.value.id))
)
```

Now you can see that another fetcher function is being called. All `.deferRel`... functions needs two arguments instead of one. We have to add the relation object as the first argument, the second is a function which will get a mapping value from entity.

We just added two relations to both `User` and `Link` object types. If you tried to run this, you would probably experience some issues. It's because now we have a circular reference in the Object type declaration. There are two things we have to do to avoid this issue:

Make `Link` and `User` lazy values. Additionally all types explicitly if you haven't done so yet:

```
lazy val UserType: ObjectType[Unit, User] =
      deriveObjectType[Unit, User]//...

lazy val LinkType: ObjectType[Unit, Link] =
      deriveObjectType[Unit, Link]//...
```

Wait, but why there is a `Unit` in type where should be our context type? Because if we don't use it explicitly inside field declaration (for example to get some data stored in context) we can do it this way. In such case our object field fits any context, not only defined one.

Now open the `graphiql` console in browser and try to execute this query: (tip: if the autocomplete doesn't work for the new fields, try to refresh the page)

```
query {
  link(id: 1){
    id
    url
    createdAt
    postedBy {
      name
      links {
        id
        url
      }
    }
  }
}
```

As you can see, both relations work perfectly.

Time to add the rest of them.

Vote - User Relation

Before I go further, try to do it yourself. All steps you need to do are similar to the those we have already done.

You have done half of the work already. There is `userId` field in the `Vote` model. Database is also prepared, there is not much work to do here.

Ok. Let's begin from proper database function.

In `DAO` class add following function:

```
def getVotesByUserIds(ids: Seq[Int]): Future[Seq[Vote]] = {
    db.run {
      Votes.filter(_.userId inSet ids).result
    }
}
```

The rest of the changes will be applied in the `GraphQLSchema` file.

Add a relation between `Vote` and `User`

```
val voteByUserRel =
      Relation[Vote, Int]("byUser", v => Seq(v.userId))
```

Don't forget in `Relation` we always have to return a sequence! Also we have to change the fetcher definition.

Change the `votesFetcher` definition with the following:

```
val votesFetcher = Fetcher.rel(
    (ctx: MyContext, ids: Seq[Int]) =>
      ctx.dao.getVotes(ids),

    (ctx: MyContext, ids: RelationIds[Vote]) =>
      ctx.dao.getVotesByUserIds(ids(voteByUserRel))

)
```

Change `UserType`:

Inside `AddField` type class, add a new field:

```
Field("votes", ListType(VoteType), resolve = c =>
votesFetcher.deferRelSeq(voteByUserRel, c.value.id))
```

Also modify the defined `VoteType`:

Replace `VoteType` with the following code:

```
lazy val VoteType: ObjectType[Unit, Vote] =
deriveObjectType[Unit, Vote](

    Interfaces(IdentifiableType),
```

```
    ExcludeFields("userId"),

    AddFields(Field("user",  UserType, resolve = c =>
usersFetcher.defer(c.value.userId)))

  )
```

That's all. After this changes you should be able to execute the query like this:

```
query {
  link(id: 1){
    id
    url
    createdAt
    postedBy {
      name
      links {
        id
        url
      }
      votes {
        id
        user {
          name
        }
      }
    }
  }
}
```

As you can see we can ask for users who vote for links posted by the author of the current link. Simple like that.

Vote - Link Relation

One relation is still missing in our example. In my opinion you have enough knowledge to try and write it yourself. After that I'll do it step by step. Reminder: case classes and database setup support this relation, you do not need to change anything there.

Lets start from defining relation object:

Add `voteByLinkRel` *constant to the* `GraphQLSchema` *file.*

```
val voteByLinkRel =
      Relation[Vote, Int]("byLink", v => Seq(v.linkId))
```

Now we can add the `votes` field to the `LinkType`.

Add the following code after the existing `ReplaceField`.

```
AddFields(
      Field("votes", ListType(VoteType), resolve = c =>
votesFetcher.deferRelSeq(voteByLinkRel, c.value.id))
)
```

You see the similarities between both `votes` fields, don't you?

```
//UserType
Field("votes", ListType(VoteType), resolve = c =>
votesFetcher.deferRelSeq(voteByUserRel, c.value.id))

//LinkType
Field("votes", ListType(VoteType), resolve = c =>
votesFetcher.deferRelSeq(voteByLinkRel, c.value.id))
```

Both are almost the same, the only difference is the type of `Relation` we're using as the first argument. Actually in this way you can add any relation you want.

Now you should be able to query for this field.

The second part won't be as easy.

Please look at the existing `votesFetcher` definition:

```
val votesFetcher = Fetcher.rel(

    (ctx: MyContext, ids: Seq[Int]) => ctx.dao.getVotes(ids),

    (ctx: MyContext, ids: RelationIds[Vote]) =>
ctx.dao.getVotesByUserIds(ids(voteByUserRel))

)
```

The first function fetches votes by their id. Nothing to comment here. The second function, on the other hand, fetches votes by relation, or more specifically, by `voteByUserRel` relation. There is no fetcher API that supports more than one relation function, so we have to refactor it a little bit.

In our case, we want to fetch votes by any relation, either with `User` or with `Link`.

`ids(voteByUserRel)` extracts the users' ids and passes those to the db function, we have to change it. It is a good idea to pass `ids` down to the function,

and in `DAO` decide which field it should use to filter.

Replace the second function of `votesFetcher` with the following one:

```
(ctx: MyContext, ids: RelationIds[Vote]) =>
ctx.dao.getVotesByRelationIds(ids)
```

There is one missing part: `DAO.getVotesByRelationIds` function, let's create it now. This function should match the kind of relation we're asking for, and filter by field depends on that relation.

To the `DAO` file add `getVotesByRelationIds` function with code:

```
//add to imports:
import sangria.execution.deferred.{RelationIds,
SimpleRelation}


//add in body
def getVotesByRelationIds(rel: RelationIds[Vote]):
Future[Seq[Vote]] =

  db.run(
    Votes.filter { vote =>
      rel.rawIds.collect({

        case (SimpleRelation("byUser"), ids: Seq[Int]) =>
vote.userId inSet ids

        case (SimpleRelation("byLink"), ids: Seq[Int]) =>
vote.linkId inSet ids

      }).foldLeft(true: Rep[Boolean])(_ || _)

    } result
  )
```

The function above uses pattern matching to recognize with type of relation it has and depends on that relation uses proper filter.

The last thing to do is to change `VoteType` definition. We have to remove `linkId` property and instead add `link` field which returns entire `Link` object.

Replace current `VoteType` declaration with the following one:

```
lazy val VoteType: ObjectType[Unit, Vote] =
deriveObjectType[Unit, Vote](

    Interfaces(IdentifiableType),

    ExcludeFields("userId", "linkId"),

    AddFields(Field("user",  UserType,  resolve = c =>
usersFetcher.defer(c.value.userId))),

    AddFields(Field("link",  LinkType,  resolve = c =>
linksFetcher.defer(c.value.linkId)))

  )
```

Now you're ready to execute a query like that:

```
query {
  links(ids :[1,2]){
    url
    votes {
      user{
        name
      }
    }
  }
}
```

You can also delete `DAO.getVotesByUserIds` function, we won't need it anymore.

Recap

We achieved our goal for this chapter, our models have new functions:

`User` has `links` and `votes` fields.

`Link` has `postedBy` and `votes` fields.

`Vote` has `user` and `link` fields.

Now we can fetch the related data...

In the next chapter you will learn how to add and save entities with GraphQL mutations.

Mutations

In the last chapters you've learnt how to use GraphQL to read data. Let's move forward, to adding data. When you want to add data, you use almost the same syntax. How does the server know when you want to write data instead of reading? You have to use the `mutation` keyword instead of `query`. That's all. Actually not all, but you will learn about the differences in this chapter.

Adding a new user

Let's start with the mutation which adds a new user to the database, like in the howtographql.com common schema

```
mutation {
  createUser(name: String!, authProvider:
AuthProviderSignupData!): User
}

input AuthProviderSignupData {
  email: AUTH_PROVIDER_EMAIL
}

input AUTH_PROVIDER_EMAIL {
  email: String!
  password: String!
}
```

It isn't hard to imagine what this mutation does. The name suggests it matches our interest - it creates an user, takes two parameters of type `String` and `AuthProviderSignupData` and returns a User in response.

But wait... until now we've been using `type` not `input`. So what is this? `input` is a type that can be used as a parameter. You will frequently see it among `mutations`.

> Let's try to implement mutation in the following order:

> Define case classes for inputs

> Define InputObjectType's for those classes,

> Define ObjectType responsible for all Mutations

> Tell a Schema to use this object.

Create case classes

Create classes needed for inputs.

```
case class AuthProviderEmail(email: String, password: String)

case class AuthProviderSignupData(email: AuthProviderEmail)
```

Define InputObjectType's

`InputObjectType` is to `input` what `ObjectType` is to the `type` keyword. It tells Sangria how to understand data. In fact you can define `ObjectType` and `InputObjectType` for the same case class, or even more than one. A good example is a `User` entity which consists of many fields. But if you need different data when you register a new user and during sign in, you can create different `InputObjectType`'s.

In GraphQLSchema.scala add the following definitions:

```
implicit val AuthProviderEmailInputType:
InputObjectType[AuthProviderEmail] =
      deriveInputObjectType[AuthProviderEmail](
   InputObjectTypeName("AUTH_PROVIDER_EMAIL")
)

lazy val AuthProviderSignupDataInputType:
InputObjectType[AuthProviderSignupData] =
      deriveInputObjectType[AuthProviderSignupData]()
```

To avoid circular dependencies of types, like we've experienced in the last chapter, we could use a `lazy` keyword for every type. But in case above, `AuthProviderEmail` is nested object in `AuthProviderSignupData` which is built by macro. Thats why we had to add `implicit`
as we have to have this nested object type in the scope in the time of macro executing.

Define Mutation Object

It will be similar to the process you already know.

In the same file add the following code:

```
  val NameArg = Argument("name", StringType)

  val AuthProviderArg = Argument("authProvider",
AuthProviderSignupDataInputType)

  val Mutation = ObjectType(
    "Mutation",
    fields[MyContext, Unit](
      Field("createUser",
        UserType,
        arguments = NameArg :: AuthProviderArg :: Nil,
        resolve = c => c.ctx.dao.createUser(c.arg(NameArg),
c.arg(AuthProviderArg))
      )
    )
  )
```

As you can see, we're missing one function in DAO

Add to DAO the following function:

```
//add to imports
import
com.howtographql.scala.sangria.models.AuthProviderSignupData

//add in body
def createUser(name: String, authProvider:
            AuthProviderSignupData): Future[User] = {

    val newUser = User(0, name, authProvider.email.email,
authProvider.email.password )


    val insertAndReturnUserQuery = (Users returning
Users.map(_.id)) into {

      (user, id) => user.copy(id = id)

    }

    db.run {
      insertAndReturnUserQuery += newUser
    }
  }
```

Add Mutation to Schema

Replace `schemaDefinition` *with the code:*

```
val SchemaDefinition = Schema(QueryType, Some(Mutation))
```

All mutations are optional so you have to wrap it in `Some`.

If you will try to run a server, you will get errors about unimplemented `FromInputs`. It's an additional step we have to do to be able to run those mutations.

Provide FromInput for input classes

Sangria needs to read a part of JSON-like structure and convert it to case classes. That's the reason why we need such `FromInput` type classes. There are two ways to do it, you can write your own mapper, but you can also use any JSON library to help with this process. In the first step we've added a dependency to the `sangria-spray-json` library, but if you want you can use any other library. Sangria uses this to convert it into proper `FromInput` type. All we need to do is to define a proper JSONReader for that case class and import some converting functions.

In the `GraphQLSchema` *file, add the following code before the definitions of InputObjectTypes:*

```
import sangria.marshalling.sprayJson._
import spray.json.DefaultJsonProtocol._

implicit val authProviderEmailFormat =
                        jsonFormat2(AuthProviderEmail)

implicit val authProviderSignupDataFormat =
                        jsonFormat1(AuthProviderSignupData)
```

Everything should work as expected now.

Test case

Perform a query in graphQL console:

```
mutation addMe {
  createUser(
    name: "Mario",
    authProvider:{
      email:{
        email:"mario@example.com",
        password:"p4ssw0rd"
      }
    }){
    id
    name
  }
}
```

Of course you can use different data :)

If everything works, we can move forward and implement two more mutations.

AddLink mutation

Implement a mutation to able run a following code:

```
createLink(description: String!, url: String!, postedById:
ID): Link
```

First try on your own, next compare to my solution.

Hint! You can skip creating case classes phase because we don't need any of them. In this case parameters uses only `String` and `Int` which are simple scalars available out-of-the-box.

In `DAO` add a function:

```
def createLink(url: String, description: String, postedBy:
Int): Future[Link] = {

  val insertAndReturnLinkQuery = (Links returning
Links.map(_.id)) into {
    (link, id) => link.copy(id = id)
  }
```

```
  db.run {

    insertAndReturnLinkQuery += Link(0, url, description,
postedBy)

  }

}
```

Also add a mutation's definition inside the `Mutation.fields` sequence.

In `GraphQLSchema` file, inside `Mutation` definition, add the following field:

```
Field("createLink",
  LinkType,
  arguments = UrlArg :: DescArg :: PostedByArg :: Nil,
  resolve = c => c.ctx.dao.createLink(c.arg(UrlArg),
c.arg(DescArg), c.arg(PostedByArg)))
```

We're missing arguments' definitions

Add arguments definitions somewhere before `Mutation`

```
val UrlArg = Argument("url", StringType)
val DescArg = Argument("description", StringType)
val PostedByArg = Argument("postedById", IntType)
```

That's all, now you should be able to run the following query:

```
mutation addLink {
  createLink(
    url: "howtographql.com",
    description: "Great tutorial page",
    postedById: 1
  ){
    url
    description
    postedBy{
      name
    }
  }

}
```

Let's implement the last mutation for voting:

Create mutation for voting

In `DAO` *add function which will be able to save new vote.*

```
def createVote(linkId: Int, userId: Int): Future[Vote] = {

  val insertAndReturnVoteQuery = (Votes returning
Votes.map(_.id)) into {
    (vote, id) => vote.copy(id = id)
  }

  db.run {
    insertAndReturnVoteQuery += Vote(0, userId, linkId)
  }
}
```

Add arguments needed by the next mutation and this mutation itself.

Add argument definitions:

```
val LinkIdArg = Argument("linkId", IntType)
val UserIdArg = Argument("userId", IntType)
```

Add mutation definition.

Add another field inside `Mutation` *objectType:*

```
Field("createVote",
  VoteType,
  arguments = LinkIdArg :: UserIdArg :: Nil,
  resolve = c => c.ctx.dao.createVote(c.arg(LinkIdArg),
c.arg(UserIdArg)))
```

We are done! You can test all those mutations in Graphiql console.

Authentication

In real life examples most of the APIs are secured. It checks whether the client has proper permissions to read/write data. In GraphQL you will do the same. I can't even imagine allowing anyone to add any data to our service anonymously.

Our goal

Provide possibility to use email and password to sign in.

Secure a query to check whether user is signed in.

The worst case scenario: The authentication/authorisation engine should support cases when the user provides the wrong credentials during sign in. Secured queries should be rejected when the user isn't signed in. We will start by providing an implementation for both cases.

Sangria's way to manage bad cases is to throw an `Exception` and catch it with the proper handler at the top level. Let's implement our cases in the suggested way.

First we have to define two exception classes in the `models` package.

```
case class AuthenticationException(message: String) extends
Exception(message)

case class AuthorizationException(message: String) extends
Exception(message)
```

`AuthenticationException` will be used during sign in, when the provided `email` and `password` values don't match the existing user.
`AuthorizationException` will be thrown when a secured query is fetched without provided credentials.

Now we have to implement a custom exception handler.

A custom `ExceptionHandler` needs a partial function which converts the type of an exception into a `HandledException`. Next this exception is internally converted into proper JSON response and sent back to the client.

In `GraphQLServer` add the following function:

```
//add to imports:
import
com.howtographql.scala.sangria.models.{AuthenticationException
, AuthorizationException}

import sangria.execution.{ExceptionHandler => EHandler, _}


//later in the body
val ErrorHandler = EHandler {

    case (_, AuthenticationException(message)) ⇒
HandledException(message)

    case (_, AuthorizationException(message)) ⇒
HandledException(message)

}
```

We've changed the name of imported `ExecutionHandler` because there is another such class in the scope, but of course you can manage this conflict in the way you prefer.

The last step and we're done.

Add this handler to our `Executor`

```
Executor.execute(
      GraphQLSchema.SchemaDefinition,
      query,
      MyContext(dao),
      variables = vars,
      operationName = operation,
      deferredResolver = GraphQLSchema.Resolver,
      exceptionHandler = ErrorHandler
    ).map(OK -> _)
      .recover {
        case error: QueryAnalysisError =>
                    BadRequest -> error.resolveError

        case error: ErrorWithResolver =>
                    InternalServerError -> error.resolveError
      }
```

Signing in

In the next step we will focus on the sign in action. But what do we need to implement it? Firstly we need an endpoint the user could use to authenticate. Next, we have to find a way to keep information whether the user is signed in correctly. At the end we have to check somehow whether the endpoint needs authorization.

FieldTag

Sangria can tag every field in queries. We could use these tags in many cases. In our example we can use a tag to check whether a field is secured. All we need is to create an object class which extends the `FieldTag` trait.

Create `Authorized` case object to check secured fields. Add to `models`

```
//add to imports:
import sangria.execution.FieldTag


//bottom of the body
case object Authorized extends FieldTag
```

Now we can tag a field. In our example we will make `addLink` mutation secured. To do so, add `tags` property with the above implemented tag.

Add `Authorized` field's tag to the `createLink` mutation field. Entire mutation's definition should look like the following one:

```
Field("createLink",
        LinkType,
        arguments = UrlArg :: DescArg :: PostedByArg :: Nil,
        tags = Authorized :: Nil,
        resolve = c => c.ctx.dao.createLink(c.arg(UrlArg),
c.arg(DescArg), c.arg(PostedByArg))),
```

The field is tagged, but Sangria won't do anything because tags are mostly informative and you have to manage the logic yourself. So it's time to implement such logic now. Assume the scenario, when the user is logged in, Sangria will keep that information and when during execution it will meet the field tagged with an `Authorised` tag, it will check whether the user is signed in.

To keep information about the user we could use the `MyContext` class. As you probably remember you can use the same context object in every subsequent query. So it perfectly fits our case.

Extend `MyContext` to keep information about the current user, with some helper function we will use later in this chapter. The `MyContext` class should look like this after changes:

```
package com.howtographql.scala.sangria


import
com.howtographql.scala.sangria.models.{AuthenticationException
, AuthorizationException, User}


import scala.concurrent._
import scala.concurrent.duration.Duration

case class MyContext(dao: DAO, currentUser: Option[User] =
None){

  def login(email: String, password: String): User = {

    val userOpt = Await.result(dao.authenticate(email,
password),   Duration.Inf)

    userOpt.getOrElse(
      throw AuthenticationException("email or password are
incorrect!")
    )

  }


  def ensureAuthenticated() =

    if(currentUser.isEmpty)

      throw AuthorizationException("You do not have
permission. Please sign in.")

}
```

The `currentUser` is a property to keep information about the signed in user. `login` function is a helper function for authorisation, it responds with user when credential fits an existing user, in the other case it will throw an exception we've defined at the beginning of this chapter. Note that I've used `Duration.Inf` you should avoid it in production code, but I wanted to keep it simple. `ensureAuthenticated` checks the `currentUser` property and throws an exception in case it's empty.

Add the `authenticate` function to the `DAO` class, to look in the database for an user with the provided credentials.

```
def authenticate(email: String, password: String):
Future[Option[User]] = db.run {

    Users.filter(u => u.email === email && u.password ===
password).result.headOption

}
```

The last step is to provide the `login` mutation.

Add the `login` mutation with the following code:

```
//before Mutation object definition:
val EmailArg = Argument("email", StringType)
val PasswordArg = Argument("password", StringType)

//in Mutation definition
Field("login",
  UserType,
  arguments = EmailArg :: PasswordArg :: Nil,
  resolve = ctx => UpdateCtx(
    ctx.ctx.login(ctx.arg(EmailArg), ctx.arg(PasswordArg))){
user =>
      ctx.ctx.copy(currentUser = Some(user))
    }
)
```

Most of the code above should be understandable by now, with the exception of resolve, which I will explain soon. `UpdateCtx` is an action which takes two parameters. The first is a function responsible for producing a response. The output of first function is passed to the second function which has to respond with a context type. This context is replaced and used in all subsequent queries. In our case I use `ctx.ctx.login(ctx.arg(EmailArg), ctx.arg(PasswordArg))` as a first function because I want to get `User` type in response. When the first function succeeds, this user will be passed to the second one and used to set the `currentUser` property.

At this point you can execute `login` mutation successfully. But `createLink` can still be accessible to anyone.

Middleware

Sangria provides a solution for middleware during execution. `Middleware` classes

are executed during query execution. If there is more than one `Middleware` class, all of them will be executed one by one. This way you can add logic which will be executed around a field or even around an entire query. The main advantage of such solution is to keep this logic completely separate from the business code. For example you can use it for benchmarking and turn it off on production environment. But in our case we will use `Middleware` to catch secured fields.

Our implementation needs to get access to the field before resolving. When the field has an `Authorized` FieldTag it should check whether the user is authenticated.

Create a file named `AuthMiddleware.scala` with the following code:

```
package com.howtographql.scala.sangria

import com.howtographql.scala.sangria.models.Authorized
import sangria.execution.{Middleware, MiddlewareBeforeField,
MiddlewareQueryContext}

import sangria.schema.Context

object AuthMiddleware extends Middleware[MyContext] with
MiddlewareBeforeField[MyContext] {

  override type QueryVal = Unit
  override type FieldVal = Unit

  override def beforeQuery(context:
MiddlewareQueryContext[MyContext, _, _]) = ()

  override def afterQuery(queryVal: QueryVal, context:
MiddlewareQueryContext[MyContext, _, _]) = ()

  override def beforeField(queryVal: QueryVal, mctx:
MiddlewareQueryContext[MyContext, _, _], ctx:
Context[MyContext, _]) = {

    val requireAuth = ctx.field.tags contains Authorized //1

    if(requireAuth) ctx.ctx.ensureAuthenticated() //2
    continue //3

  }

}
```

The main logic you can see in the `beforeField` function body. Firstly (1) it tries to read `Authorized` FieldTag and if it exists run `ensureAuthenticated` function from our context (2). If nothing bad happens Sangria will continue execution of a query (3)

The last step is to add this middleware to the executor.

`Executor` *should look as follows:*

```
Executor.execute(
      GraphQLSchema.SchemaDefinition,
      query,
      MyContext(dao),
      variables = vars,
      operationName = operation,
      deferredResolver = GraphQLSchema.Resolver,
      exceptionHandler = GraphQLSchema.ErrorHandler,
      middleware = AuthMiddleware :: Nil
    ).map//...
```

Recap

At this point we have secured the `createLink` mutation, but before that we have to login.
We can do it in the same query, like this:

```
mutation loginAndAddLink {
  login(
    email:"fred@flinstones.com",
    password:"wilmalove"
  ){
    name
  }

  createLink(
    url: "howtographql.com",
    description: "Great tutorial page",
    postedById: 2
  ){
    url
    description
    postedBy{
      name
    }
  }
}
```

You can experiment with the query above, check the response when you provide wrong email
or password, or what will happen when you'll skip entire `login` mutation.

The last words

Please remember I wanted to keep this code clear, you can extend the logic in many ways. For example you take user_id from signed in user instead of a field.

You can also use token based authentication instead of including email or password in every query.
You can also use FieldTags to check whether user has proper role to execute a query.

Do whatever you want, I just wanted to show you paths you can follow.

That's all. I hope this tutorial was useful for you and you've learnt something.

Don't forget the official documentation is always up to date and you can find there many helpful examples.

Acknowledgements

http://graphql.org

https://opensource.org/licenses/MIT

https://thenewstack.io/facebook-re-licenses-graphql/

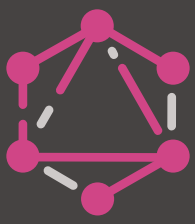
https://code.fb.com/core-data/graphql-a-data-query-language/

http://howtographql.com

https://philsturgeon.uk/api/2017/01/24/graphql-vs-rest-overview/

POWERED BY

scalac



howtographql.com