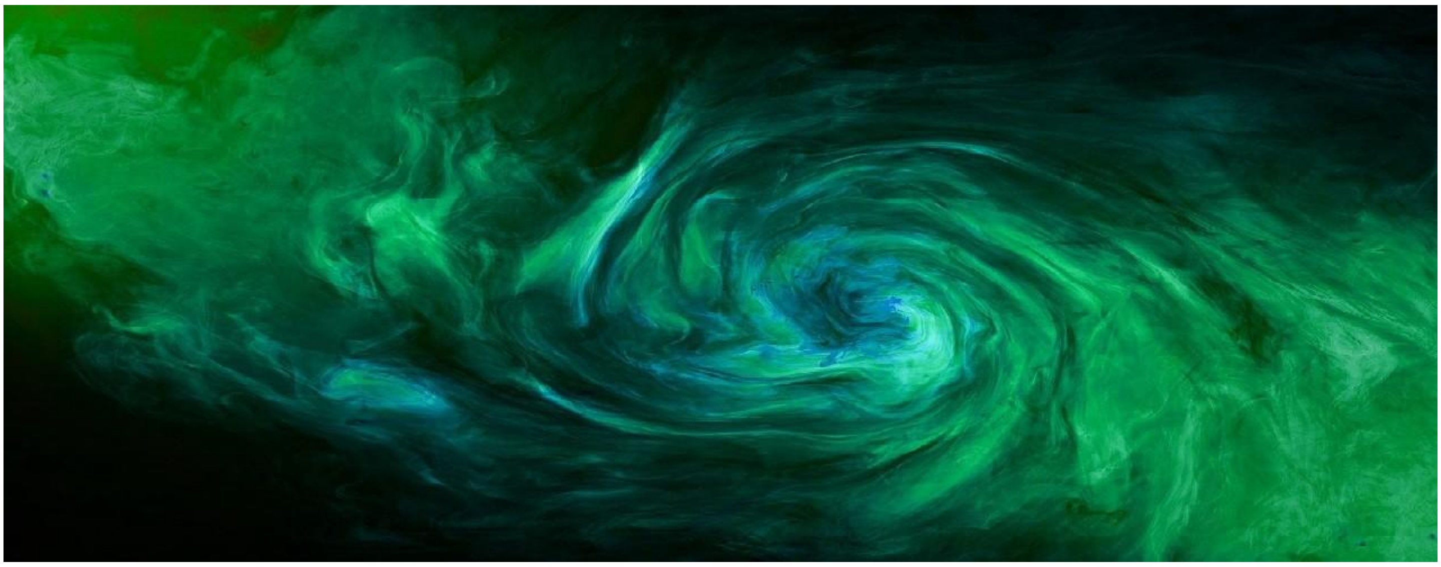
GraphQL Kotlin Tutorial with Spring Webflux

J
Julian muñoz
Dec 13, 2020 · 5 min read



In this tutorial, we are going to build a simple API with the GraphQL kotlin libraries from Expedia Group.

GraphQL kotlin provides collection of libraries to ease the development of GraphQL applications with kotlin.

GraphQL Kotlin documentation:

https://expediagroup.github.io/graphql-kotlin/docs/getting-started.html

In this example we use the Spring boot autoconfiguration library graphql-kotlin-spring-server. This library built on top of Spring Webflux, provides us with automatic schema generation, graphql-playground and out of the box support for kotlin coroutines.

Configuration

Create a Kotlin project with gradle in spring initializr:

Spring Initializr

Initializr generates spring boot project with just what you need to start quickly!

start.spring.io

This project uses the following dependencies:

```
dependencies {
        implementation("org.springframework.boot:spring-boot-starter-data-r2dbc")
        implementation("org.springframework.boot:spring-boot-starter-webflux")
        implementation("com.fasterxml.jackson.module:jackson-module-kotlin")
        implementation("io.projectreactor.kotlin:reactor-kotlin-extensions")
        implementation("org.jetbrains.kotlin:kotlin-reflect")
        implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-reactor")
        developmentOnly("org.springframework.boot:spring-boot-devtools")
        runtimeOnly("io.r2dbc:r2dbc-h2") // h2 database r2dbc driver
        runtimeOnly("com.h2database:h2") // h2 database
//      runtimeOnly("dev.miku:r2dbc-mysql") // MySQL database r2dbc driver
//      runtimeOnly("mysql:mysql-connector-java") // MySQL database
        testImplementation("org.springframework.boot:spring-boot-starter-test")
        testImplementation("io.projectreactor:reactor-test")
        implementation("com.expediagroup", "graphql-kotlin-spring-server", "3.6.8") // graphql-
}
```

build.gradle.kts hosted with ♡ by GitHub view raw

This project runs with h2 database. As reference, I also added MySQL dependencies if you to use a regular database instead or replace with your preferred database dependency, and it's r2dbc driver.

We tell graphql-kotlin to expose schema objects in the base package com.prueba.graphkt and configure the GraphQL web server. We specify the base package in the application.yml. You can take the included application.properties file located at src/main/resources and replace it with the application.yml file.

```
graphql:
  packages:
    - "com.prueba.graphkt"
```

application.yml hosted with ♡ by GitHub view raw

H2 database needs no configuration, but if you are using MySQL, set up r2dbc adding the following configuration and adjusting the configuration for your database.

```yaml
spring:
  r2dbc:
    initialization-mode: always
    url: r2dbc:mysql://localhost:3306/graphql-kt?useSSL=false&useUnicode=yes&characterEncoding=U
    username: graphql-kt-user
    password: graphql-kt-user
```

application.yml hosted with ♡ by GitHub view raw

The project uses the schema.sql, and the data.sql files in the classpath to bootstrap database data.

```sql

CREATE TABLE IF NOT EXISTS WEAPON(
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100)
);
```

schema.sql hosted with ♡ by GitHub view raw

```sql
TRUNCATE TABLE weapon;

INSERT INTO weapon (id, name) VALUES
(1, 'Whip'),
(2, 'Dagger'),
(3, 'Katana'),
(4, 'Boomerang'),
(5, 'Rapier'),
(6, 'Basilard'),
(7, 'Gurthang'),
(8, 'Marsil'),
(9, 'Claymore'),
(10,'Morning star');
```

data.sql hosted with ♡ by GitHub view raw

To load these files in the database, we have to create a db initializer class and specify of our sql files.

```kotlin
@Configuration
class DBInitializer {
    @Bean
    fun initializer(connectionFactory: ConnectionFactory): ConnectionFactoryInitializer {
        val initializer = ConnectionFactoryInitializer()
        initializer.setConnectionFactory(connectionFactory)
        val populator = CompositeDatabasePopulator()
        populator.addPopulators(ResourceDatabasePopulator(ClassPathResource("schema.sql")))
        populator.addPopulators(ResourceDatabasePopulator(ClassPathResource("data.sql")))
        initializer.setDatabasePopulator(populator)
        return initializer
    }
}
```

DBInitializer.kt hosted with ♡ by GitHub view raw

Queries

Let's get to the actual code now. We start with a basic query that fetches a mock weapon database.

```kotlin
@Component
class MockWeaponQuery(private val repository: MockWeaponRepository) : Query {

    fun mockWeapons(): List<Weapon> {
        return repository.findAll()
    }

    fun mockWeapon(id: Long): Weapon? {
        return repository.findById(id)
    }
}
```

MockWeaponQuery.kt hosted with ♡ by GitHub view raw

First we implement the query interface to create the entry point for our graphql server. Graphql-kotlin would pick the methods in our class and generate the schema for us. In this first example we have a find all function witch returns all the weapons, and a find by id function witch receives the weapon id as parameter.

Our first repository is simple, a set of mock data.

```kotlin
@Component
class MockWeaponRepository {
    fun findAll(): List<MockWeapon> = listOf(MockWeapon(1, "Rod"),MockWeapon(2, "Axe"))

    fun findById(id: Long): MockWeapon? = listOf(MockWeapon(1, "Rod"))
            .find { id == it.id }
}
```

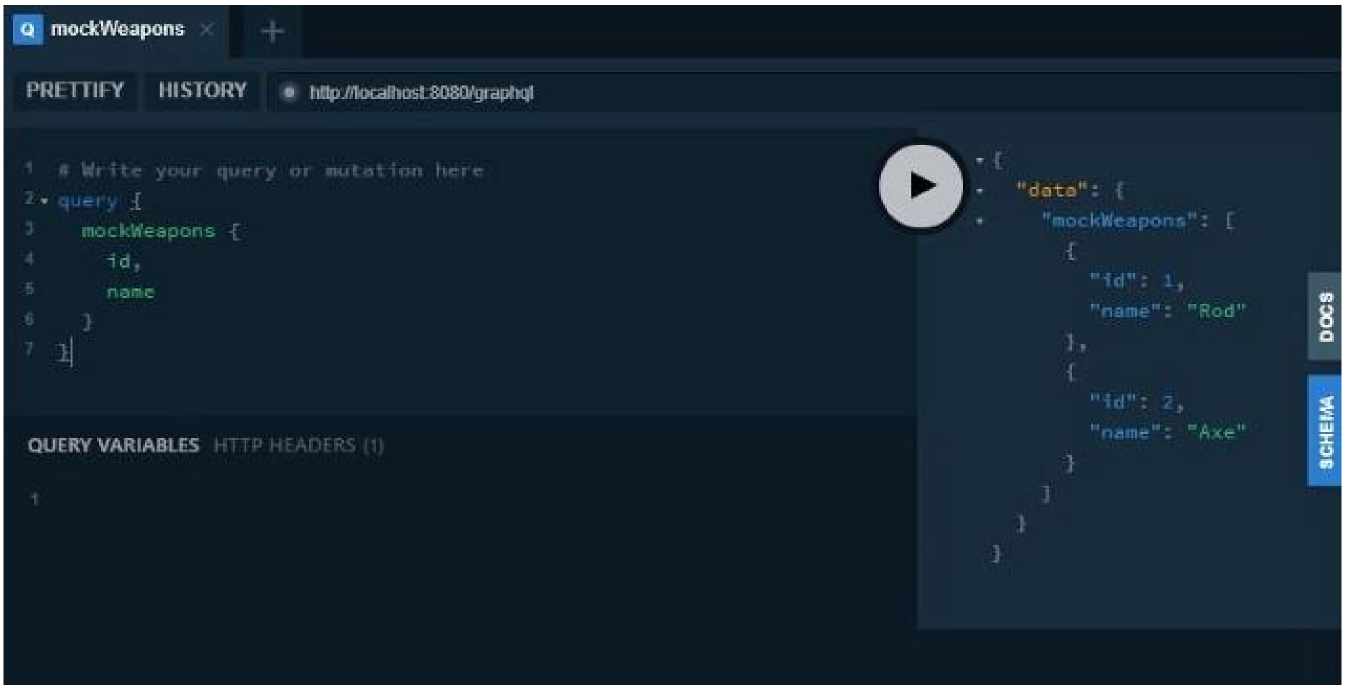
MockWeaponRepository.kt hosted with ♡ by GitHub   view raw

This is the data class.

```kotlin
data class MockWeapon(
        val id: Long,
        val name: String,
)
```

MockWeapon.kt hosted with ♡ by GitHub   view raw

Now we start the application and graphql-kotlin automatically creates the schema and the playground for us. Let's head up to localhost:8080/playground and try some queries.

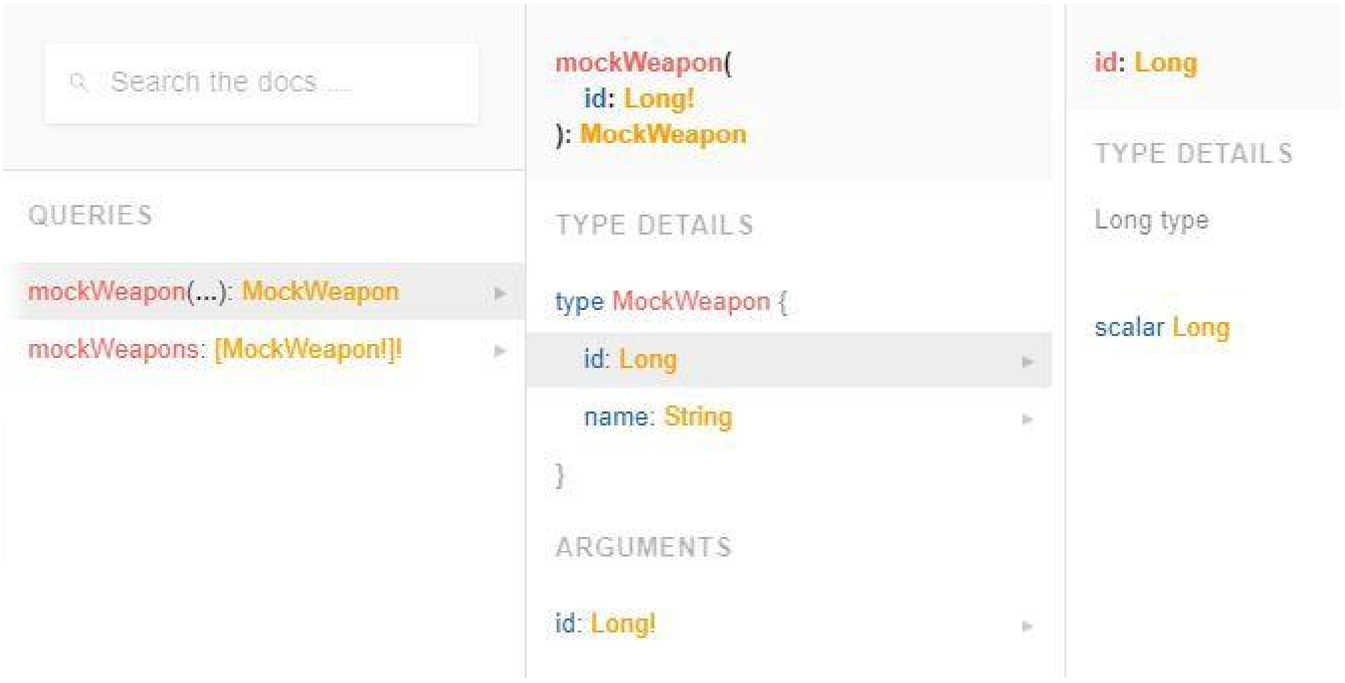


```graphql
# Write your query or mutation here
query {
  mockWeapons {
    id,
    name
  }
}
```

```json
{
  "data": {
    "mockWeapons": [
      {
        "id": 1,
        "name": "Rod"
      },
      {
        "id": 2,
        "name": "Axe"
      }
    ]
  }
}
```

We see at the left, the query to retrieve our mock weapons. We are asking our graphql server to fetch the id and the name of the weapon. There's a play button in the mid section, click it to make the query. At the right section we see the actual result of the query and two tabs.

The first tab is the documentation of the API and the second is the schema generated by graphql-kotlin. If we click the docs tab, we see a nice autogenerated API documentation.



Search the docs ...

QUERIES

mockWeapon(...): MockWeapon

mockWeapons: [MockWeapon!]!

```
mockWeapon(
  id: Long!
): MockWeapon
```

TYPE DETAILS

```
type MockWeapon {
  id: Long
  name: String
}
```

ARGUMENTS

id: Long!

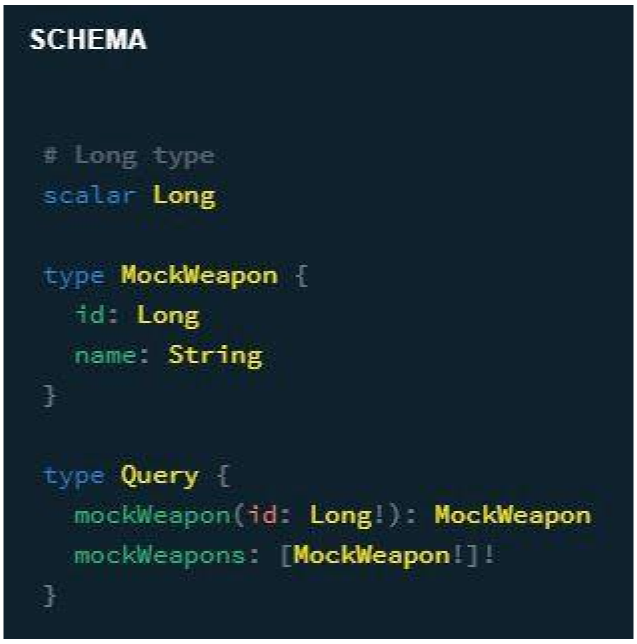
id: Long

TYPE DETAILS

Long type

scalar Long

The second tab contains our schema.



SCHEMA

```
# Long type
scalar Long

type MockWeapon {
  id: Long
  name: String
}

type Query {
  mockWeapon(id: Long!): MockWeapon
  mockWeapons: [MockWeapon!]!
}
```

As we see, graphql-kotlin has generated the graphql schema for us with just defining a query class, and a data class.

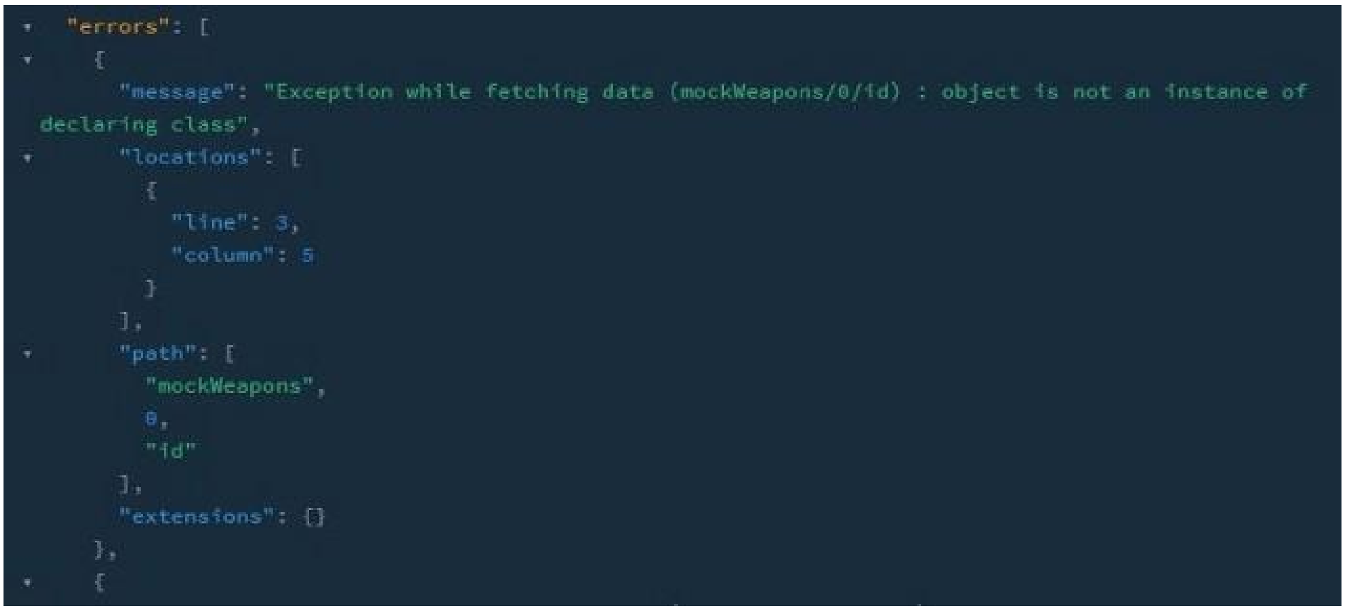
Now let's play around a little. I'm going to return a different type, List in our mockWeapons query to see what happens. This will generate an error.

```
class MockWeaponQuery(private val repository: MockWeaponRepository) : Query {

```

```kotlin
    fun mockWeapons(): List<Weapon> {
        return repository.findAll()
    }
...
}
```

MockWeaponQuery.kt hosted with ♡ by GitHub view raw

Despite our source code compiles, we get the below error.



```
"errors": [
  {
    "message": "Exception while fetching data (mockWeapons/0/id) : object is not an instance of
declaring class",
    "locations": [
      {
        "line": 3,
        "column": 5
      }
    ],
    "path": [
      "mockWeapons",
      0,
      "id"
    ],
    "extensions": {}
  },
  {
```

If we have a query returning a different type in our mock query, graphql-kotlin would throw a reflection error, as this library uses kotlin reflection to generate the schema matching the query name with the return type name.

Finally, we revert our source code to fix the error.

```kotlin
class MockWeaponQuery(private val repository: MockWeaponRepository) : Query {

    fun mockWeapons(): List<MockWeapon> {
        return repository.findAll()
    }
...
}
```

MockWeaponQuery.kt hosted with ♡ by GitHub view raw

Now we saw the basics of graphql-kotlin, we are going to a more realistic example using the actual Spring Webflux this library runs on.

Webflux and coroutines

In this section we are working with a more realistic setup, fetching data from the database using the Webflux ReactiveCrudRepository.

```kotlin
@Repository
interface WeaponRepository : ReactiveCrudRepository<Weapon, Long> {
}
```

WeaponRepository.kt hosted with ♡ by GitHub view raw

Our data class is the same as the mock version.

```kotlin
import org.springframework.data.annotation.Id

data class Weapon(
        @Id
        val id: Long,
        val name: String,
)
```

Weapon.kt hosted with ♡ by GitHub view raw

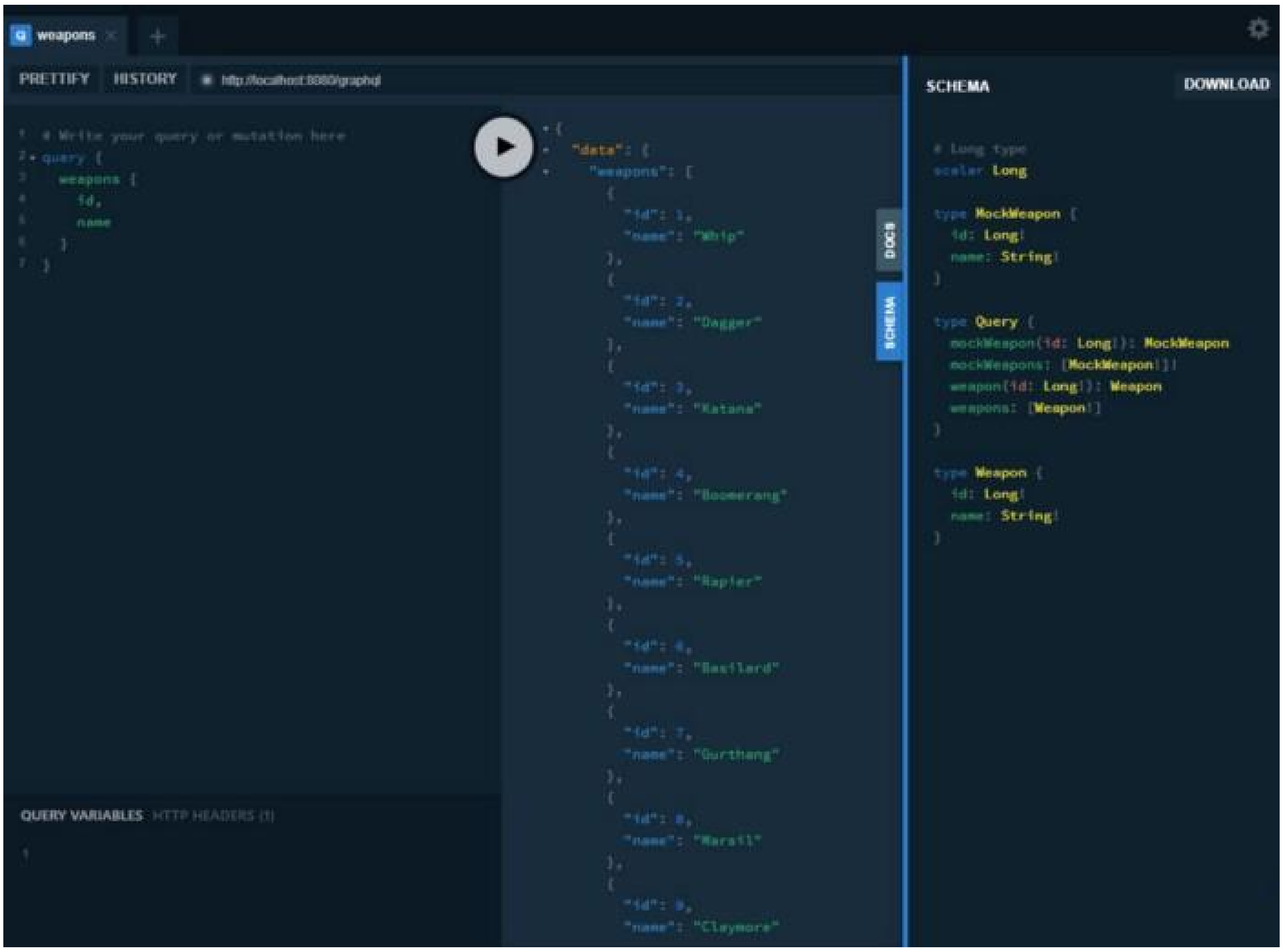
Notice the @Id annotation needed for Spring Data in order to map our data classes to database records. Now let's see the query class.

```kotlin
@Component
class WeaponQuery(private val weaponRepository: WeaponRepository) : Query {

    suspend fun weapons(): MutableList<Weapon>? {
        return weaponRepository.findAll().collectList().awaitFirst()
    }

    suspend fun weapon(id: Long): Weapon? {
        return weaponRepository.findById(id).awaitSingle()
    }
}
```

WeaponQuery.kt hosted with ♡ by GitHub view raw

We use the Webflux ReactiveCrudRepository findAll and findById stock methods to fetch our weapons, then we use the Reactor's kotlin coroutines api to transform the resulting Flux to a suspend function. Kotlin coroutines have built-in support for graphql-kotlin-schema-generator, if your are using other asynchronous models like RxJava and Reactor, you have to specify your own data fetcher. Form more info about other asynchronous models, follow this link https://expediagroup.github.io/graphql-kotlin/docs/schema-generator/execution/async-models.

Now we have finish our graphql API, let's try some queries on graphql playground.

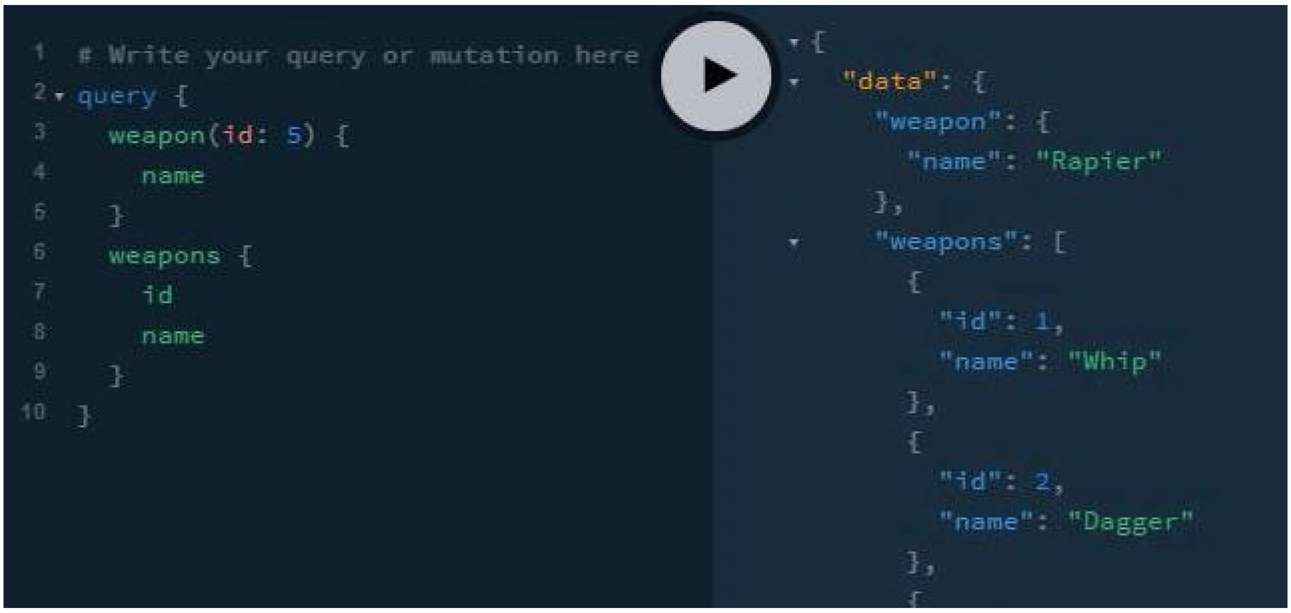


```
# Write your query or mutation here
query {
  weapons {
    id,
    name
  }
}
```

```
{
  "data": {
    "weapons": [
      {
        "id": 1,
        "name": "Whip"
      },
      {
        "id": 2,
        "name": "Dagger"
      },
      {
        "id": 3,
        "name": "Katana"
      },
      {
        "id": 4,
        "name": "Boomerang"
      },
      {
        "id": 5,
        "name": "Rapier"
      },
      {
        "id": 6,
        "name": "Basilard"
      },
      {
        "id": 7,
        "name": "Gurthang"
      },
      {
        "id": 8,
        "name": "Narsil"
      },
      {
        "id": 9,
        "name": "Claymore"
```

```
# Long type
scalar Long

type MockWeapon {
  id: Long!
  name: String!
}

type Query {
  mockWeapon(id: Long!): MockWeapon
  mockWeapons: [MockWeapon!]!
  weapon(id: Long!): Weapon
  weapons: [Weapon!]
}

type Weapon {
  id: Long!
  name: String!
}
```

We are querying now data from our database and have the full schema. Let's try some more queries.



```
# Write your query or mutation here
query {
  weapon(id: 5) {
    name
  }
  weapons {
    id
    name
  }
}
```

```
{
  "data": {
    "weapon": {
      "name": "Rapier"
    },
    "weapons": [
      {
        "id": 1,
        "name": "Whip"
      },
      {
        "id": 2,
        "name": "Dagger"
      },
      {
```

```
    "id": 3,
    "name": "Katana"
  },
  {
    "id": 4,
    "name": "Boomerang"
  },
```

There you go, a fully working graphql server easily implemented with Spring boot and graphql-kotlin.

We have a lot of flexibility for defining our queries with this setup, for example, we can arrange our application in three layer architecture. For a tree layer architecture setup, we can move the query class logic to a service.

```kotlin
@Service
class WeaponService(private var weaponRepository: WeaponRepository) {
    suspend fun findAll(): List<Weapon> {
        return weaponRepository.findAll().collectList().awaitFirstOrDefault(listOf())
    }

    suspend fun findById(id: Long): Weapon? {
        return weaponRepository.findById(id).awaitSingle()
    }
}
```

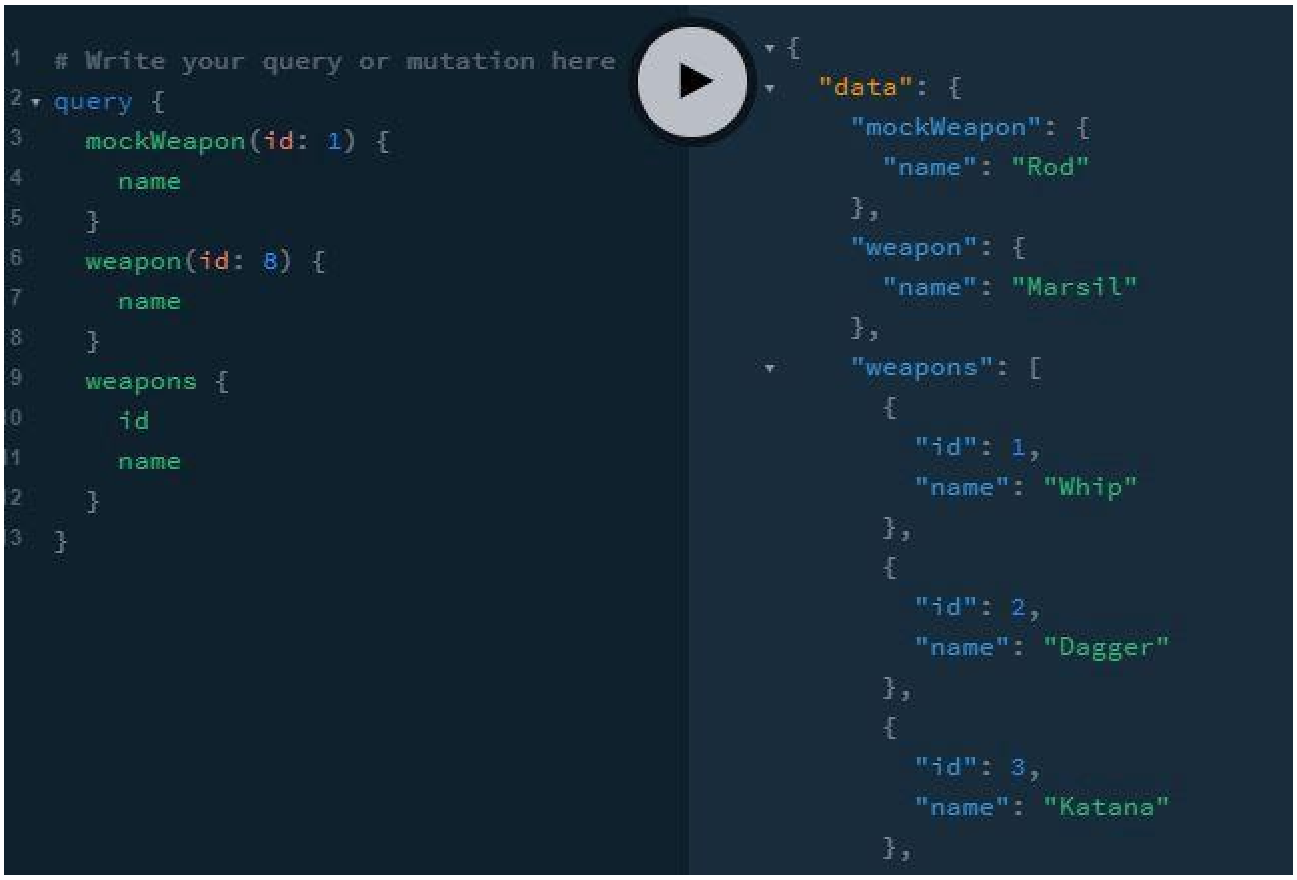
WeaponService.kt hosted with ♡ by GitHub view raw

Our query class be the following.

```kotlin
@Component
class WeaponQuery(private val weaponService: WeaponService) : Query {

    suspend fun weapons(): List<Weapon> {
        return weaponService.findAll()
    }

    suspend fun weapon(id: Long): Weapon? {
        return weaponService.findById(id)
    }
}
```

WeaponQuery.kt hosted with ♡ by GitHub view raw

The application is still working as expected.



```
# Write your query or mutation here
query {
  mockWeapon(id: 1) {
    name
  }
  weapon(id: 8) {
    name
  }
  weapons {
    id
    name
  }
}
```

```
{
  "data": {
    "mockWeapon": {
      "name": "Rod"
    },
    "weapon": {
      "name": "Marsil"
    },
    "weapons": [
      {
        "id": 1,
        "name": "Whip"
      },
      {
        "id": 2,
        "name": "Dagger"
      },
      {
        "id": 3,
        "name": "Katana"
      },
```

That's it for this tutorial.

Conclusions

We created a graphql server with graphql-kotlin. The implementation was easy as we only have to provide the data with the help of kotlin coroutines. We can have Spring boot application as complex as we need and allow our clients to decide on how they fetch the API. As long as the query name matches the resulting entity name. The next steps is learning how to make paginated queries with the help of Spring data, avoid DDOS attacks because of recursive queries and to secure our API with Spring security.

For more information on GraphQL Kotlin check the library documentation and examples: https://expediagroup.github.io/graphql-kotlin.

Link of finished tutorial repo:

julianmunozm45/graphql-kt-tutorial

This an example project for the Graphql-kotlin library: https://expediagroup.github.io/graphql-kotlin Graphql-kotlin...

github.com

Originally published at http://github.com.

Kotlin GraphQL Spring Webflux Spring Boot Kotlin Coroutines