Quarkus - GraphQL

This guide demonstrates how your Quarkus application can utilize the **Eclipse MicroProfile GraphQL** specification through the SmallRye GraphQL extension.

As the GraphQL specification website states:

> GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

GraphQL was originally developed by **Facebook** in 2012 and has been an open standard since 2015.

GraphQL is not a replacement for REST API specification but merely an alternative. Unlike REST, GraphQL API's have the ability to benefit the client by:

Preventing Over-fetching and Under-fetching

REST API's are server-driven fixed data responses that cannot be determined by the client. Although the client does not require all the fields the client must retrieve all the data hence `Over-fetching`. A client may also require multiple REST API calls according to the first call (HATEOAS) to retrieve all the data that is required thereby `Under-fetching`.

API Evolution

Since GraphQL API's returns data that are requested by the client adding additional fields and capabilities to existing API will not create breaking changes to existing clients.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3

Architecture

In this guide, we build a simple GraphQL application that exposes a GraphQL API at `/graphql`.

This example was inspired by a popular GraphQL API.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `microprofile-graphql-quickstart` [directory](#).

Creating the Maven Project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=microprofile-graphql-quickstart \
    -DclassName="org.acme.microprofile.graphql.FilmResource" \
    -Dextensions="graphql"
cd microprofile-graphql-quickstart
```

This command generates a Maven project, importing the `smallrye-graphql` extension which is an implementation of the MicroProfile GraphQL specification used in Quarkus.

If you already have your Quarkus project configured, you can add the `smallrye-graphql` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="graphql"
```

This will add the following to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-graphql</artifactId>
</dependency>
```

Preparing an Application: GraphQL API

In this section we will start creating the GraphQL API.

First, create the following entities representing a film from a galaxy far far away:

```
package org.acme.microprofile.graphql;
```

```java
public class Film {

    private String title;
    private Integer episodeID;
    private String director;
    private LocalDate releaseDate;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Integer getEpisodeID() {
        return episodeID;
    }

    public void setEpisodeID(Integer episodeID) {
        this.episodeID = episodeID;
    }

    public String getDirector() {
        return director;
    }

    public void setDirector(String director) {
        this.director = director;
    }

    public LocalDate getReleaseDate() {
        return releaseDate;
    }

    public void setReleaseDate(LocalDate releaseDate) {
        this.releaseDate = releaseDate;
    }

}

public class Hero {

    private String name;
    private String surname;
    private Double height;
    private Integer mass;
    private Boolean darkSide;
    private LightSaber lightSaber;
```

```java
    private List<Integer> episodeIds = new ArrayList<>();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public Double getHeight() {
        return height;
    }

    public void setHeight(Double height) {
        this.height = height;
    }

    public Integer getMass() {
        return mass;
    }

    public void setMass(Integer mass) {
        this.mass = mass;
    }

    public Boolean getDarkSide() {
        return darkSide;
    }

    public void setDarkSide(Boolean darkSide) {
        this.darkSide = darkSide;
    }

    public LightSaber getLightSaber() {
        return lightSaber;
    }

    public void setLightSaber(LightSaber lightSaber) {
        this.lightSaber = lightSaber;
    }
```

```
    public List<Integer> getEpisodeIds() {
        return episodeIds;
    }

    public void setEpisodeIds(List<Integer> episodeIds) {
        this.episodeIds = episodeIds;
    }
}

enum LightSaber {
    RED, BLUE, GREEN
}
```

The classes we have just created describe the GraphQL schema which is a set of possible data (objects, fields, relationships) that a client can access.

Let's continue with an example CDI bean, that would work as a repository:

```
@ApplicationScoped
public class GalaxyService {

    private List<Hero> heroes = new ArrayList<>();

    private List<Film> films = new ArrayList<>();

    public GalaxyService() {

        Film aNewHope = new Film();
        aNewHope.setTitle("A New Hope");
        aNewHope.setReleaseDate(LocalDate.of(1977, Month.MAY, 25));
        aNewHope.setEpisodeID(4);
        aNewHope.setDirector("George Lucas");

        Film theEmpireStrikesBack = new Film();
        theEmpireStrikesBack.setTitle("The Empire Strikes Back");
        theEmpireStrikesBack.setReleaseDate(LocalDate.of(1980,
Month.MAY, 21));
        theEmpireStrikesBack.setEpisodeID(5);
        theEmpireStrikesBack.setDirector("George Lucas");

        Film returnOfTheJedi = new Film();
        returnOfTheJedi.setTitle("Return Of The Jedi");
        returnOfTheJedi.setReleaseDate(LocalDate.of(1983,
Month.MAY, 25));
        returnOfTheJedi.setEpisodeID(6);
        returnOfTheJedi.setDirector("George Lucas");

        films.add(aNewHope);
        films.add(theEmpireStrikesBack);
```

```java
        films.add(returnOfTheJedi);

        Hero luke = new Hero();
        luke.setName("Luke");
        luke.setSurname("Skywalker");
        luke.setHeight(1.7);
        luke.setMass(73);
        luke.setLightSaber(LightSaber.GREEN);
        luke.setDarkSide(false);
        luke.getEpisodeIds().addAll(Arrays.asList(4, 5, 6));

        Hero leia = new Hero();
        leia.setName("Leia");
        leia.setSurname("Organa");
        leia.setHeight(1.5);
        leia.setMass(51);
        leia.setDarkSide(false);
        leia.getEpisodeIds().addAll(Arrays.asList(4, 5, 6));


        Hero vader = new Hero();
        vader.setName("Darth");
        vader.setSurname("Vader");
        vader.setHeight(1.9);
        vader.setMass(89);
        vader.setDarkSide(true);
        vader.setLightSaber(LightSaber.RED);
        vader.getEpisodeIds().addAll(Arrays.asList(4, 5, 6));

        heroes.add(luke);
        heroes.add(leia);
        heroes.add(vader);

    }

    public List<Film> getAllFilms() {
        return films;
    }

    public Film getFilm(int id) {
        return films.get(id);
    }

    public List<Hero> getHeroesByFilm(Film film) {
        return heroes.stream()
                .filter(hero ->
 hero.getEpisodeIds().contains(film.getEpisodeID()))
                .collect(Collectors.toList());
    }
```

```
    public void addHero(Hero hero) {
        heroes.add(hero);
    }

    public Hero deleteHero(int id) {
        return heroes.remove(id);
    }

    public List<Hero> getHeroesBySurname(String surname) {
        return heroes.stream()
                .filter(hero -> hero.getSurname().equals(surname))
                .collect(Collectors.toList());
    }
}
```

Now, let's create our first GraphQL API.

Edit the `org.acme.microprofile.graphql.FilmResource` class as following:

```
@GraphQLApi  ①
public class FilmResource {

    @Inject
    GalaxyService service;

    @Query("allFilms")  ②
    @Description("Get all Films from a galaxy far far away")  ③
    public List<Film> getAllFilms() {
        return service.getAllFilms();
    }
}
```

① `@GraphQLApi` annotation indicates that the CDI bean will be a GraphQL endpoint

② `@Query` annotation defines that this method will be queryable with the name `allFilms`

③ Documentation of the queryable method

> The value of the `@Query` annotation is optional and would implicitly be defaulted to the method name if absent.

This way we have created our first queryable API which we will later expand.

Launch

Launch the quarkus app:

```
./mvnw compile quarkus:dev
```

Introspect

The full schema of the GraphQL API can be retrieved by calling the following:

```
curl http://localhost:8080/graphql/schema.graphql
```

The server will return the complete schema of the GraphQL API.
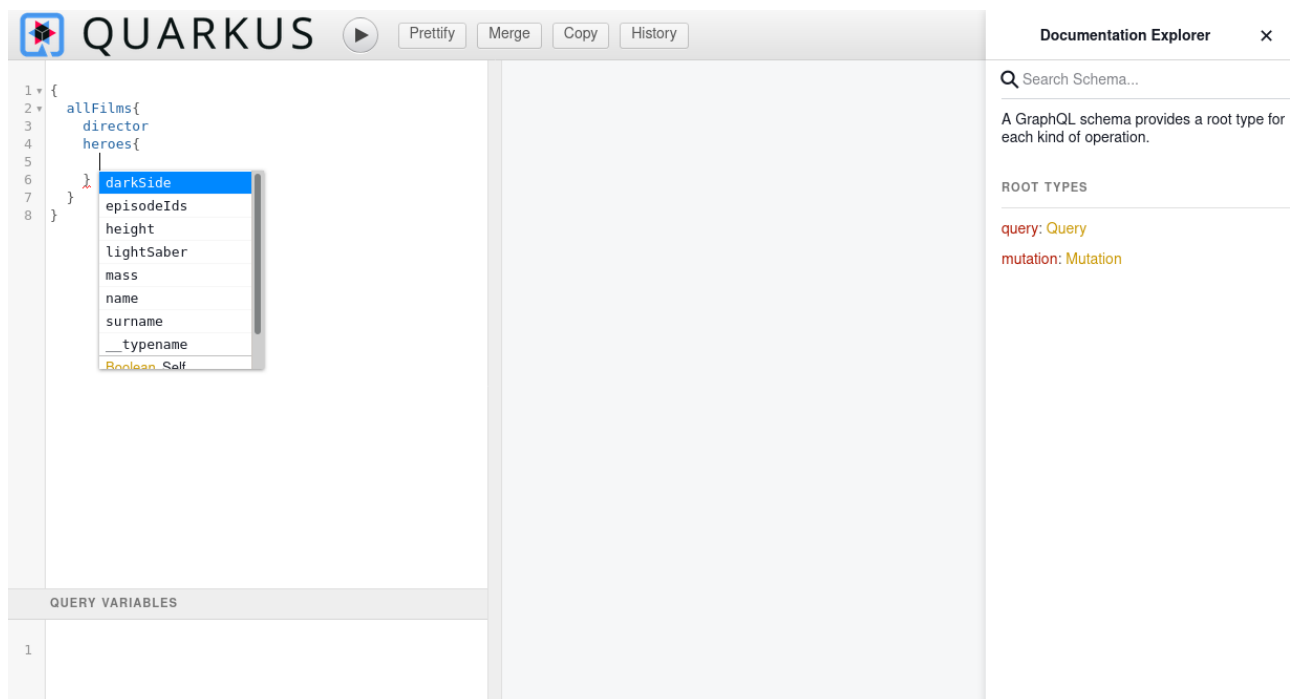
GraphiQL UI

ℹ   Experimental - not included in the MicroProfile specification

GraphiQL UI is a great tool permitting easy interaction with your GraphQL APIs.

The Quarkus `smallrye-graphql` extension ships with `GraphiQL` and enables it by default in `dev` and `test` modes, but it can also be explicitly configured for `production` mode as well.

GraphiQL can be accessed from http://localhost:8080/graphql-ui/ .

Query the GraphQL API

Now visit the GraphiQL page that has been deployed in `dev` mode.

Enter the following query to GraphiQL and press the `play` button:

```
query allFilms {
  allFilms {
    title
    director
    releaseDate
    episodeID
  }
}
```

Since our query contains all the fields in the `Film` class we will retrieve all the fields in our response. Since GraphQL API responses are client determined, the client can choose which fields it will require.

Let's assume that our client only requires `title` and `releaseDate` making the previous call to the API `Over-fetching` of unnecessary data.

Enter the following query into GraphiQL and hit the `play` button:

```
query allFilms {
  allFilms {
    title
    releaseDate
  }
}
```

Notice in the response we have only retrieved the required fields. Therefore, we have prevented `Over-fetching`.

Let's continue to expand our GraphQL API by adding the following to the `FilmResource` class.

```
    @Query
    @Description("Get a Films from a galaxy far far away")
    public Film getFilm(@Name("filmId") int id) {
        return service.getFilm(id);
    }
```

⚠️ Notice how we have excluded the value in the `@Query` annotation. Therefore, the name of the query is implicitly set as the method name excluding the `get`.

This query will allow the client to retrieve the film by id.

Enter the following into `GraphiQL` and make a request.

```
query getFilm {
  film(filmId: 1) {
    title
    director
    releaseDate
    episodeID
  }
}
```

The `film` query method requested fields can be determined as such in our previous example. This way we can retrieve individual film information.

However, say our client requires both films with filmId `0` and `1`. In a REST API the client would have to make two calls to the API. Therefore, the client would be `Under-fetching`.

In GraphQL it is possible to make multiple queries at once.

Enter the following into GraphiQL to retrieve two films:

```
query getFilms {
  film0: film(filmId: 0) {
    title
    director
    releaseDate
    episodeID
  }
  film1: film(filmId: 1) {
    title
    director
    releaseDate
    episodeID
  }
}
```

This enabled the client to fetch the required data in a single request.

Expanding the API

Until now, we have created a GraphQL API to retrieve film data. We now want to enable the clients to retrieve the `Hero` data of the `Film`.

Add the following to our `FilmResource` class:

```
    public List<Hero> heroes(@Source Film film) { ①
        return service.getHeroesByFilm(film);
    }
```

① Enable `List<Hero>` data to be added to queries that respond with `Film`

By adding this method we have effectively changed the schema of the GraphQL API. Although the schema has changed the previous queries will still work. Since we only expanded the API to be able to retrieve the `Hero` data of the `Film`.

Enter the following into GraphiQL to retrieve the film and hero data.

```
query getFilmHeroes {
  film(filmId: 1) {
    title
    director
    releaseDate
    episodeID
    heroes {
      name
      height
      mass
      darkSide
      lightSaber
    }
  }
}
```

The response now includes the heroes of the film.

Mutations

Mutations are used when data is created, updated or deleted.

Let's now add the ability to add and delete heroes to our GraphQL API.

Add the following to our `FilmResource` class:

```
    @Mutation
    public Hero createHero(Hero hero) {
        service.addHero(hero);
        return hero;
    }

    @Mutation
    public Hero deleteHero(int id) {
        return service.deleteHero(id);
    }
```

Enter the following into GraphiQL to insert a Hero:

```
mutation addHero {
  createHero(hero: {
      name: "Han",
      surname: "Solo"
      height: 1.85
      mass: 80
      darkSide: false
      episodeIds: [4, 5, 6]
    }
  )
  {
    name
    surname
  }
}
```

By using this mutation we have created a Hero entity in our service.

Notice how in the response we have retrieved the name and surname of the created Hero. This is because we selected to retrieve these fields in the response within the { } in the mutation query. This can easily be a server side generated field that the client may require.

Let's now try deleting an entry:

```
mutation DeleteHero {
  deleteHero(id :3){
    name
    surname
  }
}
```

Similar to the createHero mutation method we also retrieve the name and surname of the hero we

have deleted which is defined in `{  }`.

Creating Queries by fields

Queries can also be done on individual fields. For example, let's create a method to query heroes by their last name.

Add the following to our `FilmResource` class:

```
    @Query
    public List<Hero>
 getHeroesWithSurname(@DefaultValue("Skywalker") String surname) {
        return service.getHeroesBySurname(surname);
    }
```

By using the `@DefaultValue` annotation we have determined that the surname value will be `Skywalker` when the parameter is not provided.

Test the following queries with GraphiQL:

```
query heroWithDefaultSurname {
  heroesWithSurname{
    name
    surname
    lightSaber
  }
}
query heroWithSurnames {
  heroesWithSurname(surname: "Vader") {
    name
    surname
    lightSaber
  }
}
```

Context

You can get information about the GraphQL request anywhere in your code, using this experimental, SmallRye specific feature:

```
 @Inject
 Context context;
```

The context object allows you to get:

- the original request (Query/Mutation)

- the arguments

- the path

- the selected fields

- any variables

This allows you to optimize the downstream queries to the datastore.

See the JavaDoc for more details.

GraphQL-Java

This context object also allows you to fall down to the underlying graphql-java features by using the leaky abstraction:

```
DataFetchingEnvironment dfe =
context.unwrap(DataFetchingEnvironment.class);
```

You can also get access to the underlying `graphql-java` during schema generation, to add your own features directly:

```
public GraphQLSchema.Builder addMyOwnEnum(@Observes
GraphQLSchema.Builder builder) {

    // Here add your own features directly, example adding an Enum
    GraphQLEnumType myOwnEnum = GraphQLEnumType.newEnum()
            .name("SomeEnum")
            .description("Adding some enum type")
            .value("value1")
            .value("value2").build();

    return builder.additionalType(myOwnEnum);
}
```

By using the `@Observer` you can add anything to the Schema builder.

Map to Scalar

Another SmallRye specific experimental feature, allows you to map an existing scalar (that is mapped by the implementation to a certain Java type) to another type, or to map complex object, that would typically create a `Type` or `Input` in GraphQL, to an existing scalar.

Mapping an existing Scalar to another type:

```
public class Movie {

    @ToScalar(Scalar.Int.class)
    Long idLongThatShouldChangeToInt;

    // ....
}
```

Above will map the `Long` java type to an `Int` Scalar type, rather than the `default` `BigInteger`.

Mapping a complex object to a Scalar type:

```
public class Person {

    @ToScalar(Scalar.String.class)
    Phone phone;

    // ....
}
```

This will, rather than creating a `Type` or `Input` in GraphQL, map to a String scalar.

To be able to do the above, the `Phone` object needs to have a constructor that takes a String (or `Int` / `Date` / etc.), or have a setter method for the String (or `Int` / `Date` / etc.), or have a `fromString` (or `fromInt` / `fromDate` - depending on the Scalar type) static method.

For example:

```
public class Phone {

    private String number;

    // Getters and setters....

    public static Phone fromString(String number) {
        Phone phone = new Phone();
        phone.setNumber(number);
        return phone;
    }
}
```

See more about the `@ToScalar` feature in the JavaDoc.

Error code

You can add an error code on the error output in the GraphQL response by using the (SmallRye specific) @ErrorCode:

```
@ErrorCode("some-business-error-code")
public class SomeBusinessException extends RuntimeException {
    // ...
}
```

When SomeBusinessException occurs, the error output will contain the Error code:

```
{
    "errors": [
        {
            "message": "Unexpected failure in the system. Jarvis is working to fix it.",
            "locations": [
                {
                    "line": 2,
                    "column": 3
                }
            ],
            "path": [
                "annotatedCustomBusinessException"
            ],
            "extensions": {
                "exception":
"io.smallrye.graphql.test.apps.error.api.ErrorApi$AnnotatedCustomBusinessException",
                "classification": "DataFetchingException",
                "code": "some-business-error-code" ①
            }
        }
    ],
    "data": {
        ...
    }
}
```

① The error code

Conclusion

MicroProfile GraphQL enables clients to retrieve the exact data that is required preventing Over-

`fetching` and `Under-fetching`.

The GraphQL API can be expanded without breaking previous queries enabling easy API `evolution`.

Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.smallrye-graphql.root-path` The rootPath under which queries will be served. Default to /graphql	string	`/graph ql`
🔒 `quarkus.smallrye-graphql.metrics.enabled` Enable metrics	boolean	`false`
UI configuration	**Type**	**Default**
🔒 `quarkus.smallrye-graphql.ui.root-path` The path where GraphQL UI is available. The value `/` is not allowed as it blocks the application from serving anything else.	string	`/graph ql-ui`
🔒 `quarkus.smallrye-graphql.ui.always-include` Always include the UI. By default this will only be included in dev and test. Setting this to true will also include the UI in Prod	boolean	`false`
🔒 `quarkus.smallrye-graphql.ui.enable` If GraphQL UI should be enabled. By default, GraphQL UI is enabled.	boolean	`true`