GraphQL for MicroProfile

Jean-Francois James, Phillip Krüger, Andy McCright, Jean-Baptiste Roux, Bojan Tomic, Adam Anderson

1.0-M3, August 23, 2019

Table of Contents

Specification: GraphQL for MicroProfile

Version: 1.0-M3

Status: Draft

Release: August 23, 2019

MicroProfile GraphQL

Introduction to MicroProfile GraphQL

About GraphQL

GraphQL is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. GraphQL interprets strings from the client, and returns data in an understandable, predictable, pre-defined manner. GraphQL is an alternative, though not necessarily a replacement for REST.

GraphQL was developed internally by Facebook in 2012 before being publicly released in 2015. Facebook delivered both a specification and a reference implementation in JavaScript.

On 7 November 2018, Facebook moved the GraphQL project to the newly-established GraphQL foundation, hosted by the non-profit Linux Foundation. This is a significant milestone in terms of industry and community adoption. GraphQL is widely used by many customers.

- More info: https://en.wikipedia.org/wiki/GraphQL
- Home page: https://graphql.org/
- Specification: https://facebook.github.io/graphql/draft/

Why GraphQL

The main reasons for using GraphQL are:

- Avoiding over-fetching or under-fetching data. Clients can retrieve several types of data in a single request or can limit the response data based on specific criteria.
- Enabling data models to evolve. Changes to the schema can be made so as to not require changes on existing clients, and vice versa - this can be done without a need for a new version of the application.
- Advanced developer experience:
  - The schema defines how the data can be accessed and serves as the contract between the client and the server. Development teams on both sides can work without further communication.
  - Native schema introspection enables users to discover APIs and to refine the queries on the client-side. This advantage is increased with graphical tools such as GraphiQL and GraphQL Voyager enabling smooth and easy API discovery.
  - On the client-side, the query language provides flexibility and efficiency enabling developers to adapt to the constraints of their technical environments while reducing server round-trips.

GraphQL and REST

GraphQL and REST have many similarities and are both widely used in modern microservice applications. The two technologies also have some differences.

REST stands for "Representational State Transfer". It is an architectural style for network-based software specified by Roy Fielding in 2000 in a dissertation defining 6 theoretical constraints:

1. uniform interface
2. stateless
3. client-server
4. cacheable
5. layered system
6. code on demand (optional).

REST is often implemented as JSON over HTTP, but REST is fundamentally technically agnostic to data type and transport; it is an architectural style. In particular, it doesn't require to use HTTP. However, it recommends using the maximum capacity of the underlying network protocol to apply the 6 basic principles. For instance, REST implementations can utilize HTTP semantics with a proper use of verbs (POST, GET, PUT, PATCH, DELETE) and response codes (2xx, 4xx, 5xx).

GraphQL takes its roots from a Facebook specification published in 2015. As of this date, GraphQL has been subject to 5 releases:

- June 2018
- October 2016
- April 2016
- October 2015
- July 2015

According to it's definition: "GraphQL is a query language for describing the capabilities and requirements of data models for client-server applications."

Like REST, GraphQL is independent from particular transport protocols or data models:

- it does not endorse the use of HTTP though in practice, and like REST, it is clearly the most widely used protocol,
- it is not tied to any specific database technology or storage engine and is instead backed by existing code and data.

What make GraphQL different?

In practice, here are the main differentiating features of GraphQL compared to REST:

- **schema-driven**: a GraphQL API natively exposes a schema describing the structure of the data and operations (queries and mutations) exposed. This schema acts as a contract between the server and its clients. In a way GraphQL provides an explicit answer to the API discovery problem where REST relies on the ability of developers to properly use other mechanisms such as HATEOS and/or OpenAPI,
- **single HTTP endpoint**: a typical GraphQL API is made of a single endpoint and access to data

and operations is achieved through the query language. In a HTTP context, the endpoint is defined as a URL and the query can be transported as a query string (GET request) or in the request body (POST request),

- **flexible data retrieval**: by construction the query language enables the client to select the expected data in the response with a fine level of granularity, thus avoiding over- or under-fetching data,
- **reduction of server requests**: the language allows the client to aggregate the expected data into a single request,
- **easier version management**: thanks to the native capabilities to create new data while deprecating old ones,
- **partial results**: partial results are delivered by the GraphQL server in case of errors. A GraphQL result is made of data and errors. Clients are responsible for processing the partial results,
- **low coupling with HTTP**: GraphQL does not try to make the most of HTTP semantics. Queries can be made using GET or POST requests. The HTTP result code does not reflect the GraphQL response,
- **challenging authorization handling**: an appropriate data access authorization policy must be defined and implemented to counter the extreme flexibility of the query language. For example, one client may be authorized to access some data that others are not,
- **challenging API management**: most API management solutions are based on REST capabilities and allow for endpoint (URL-based) policies to be established. GraphQL API has a single entry point. It may be necessary to analyze the client request data to ensure it conforms to established policies. For example, it may be necessary to validate mutations or to prevent the client from executing an overly complex request that would crash the server.

GraphQL and Databases

GraphQL is about data query and manipulation but it is not a database technology:

- It is a query language for APIs,
- It is database and storage agnostic,
- It can be used in front of any kind of backend, with or without a database.

One of GraphQL's strength is its multi-datasource capability enabling a single endpoint to aggregate data from various sources with a single API.

MicroProfile GraphQL

The intent of the MicroProfile GraphQL specification is provide a "code-first" set of APIs that will enable users to quickly develop portable GraphQL-based applications in Java.

GraphQL Entities

Entities are the objects used in GraphQL. They can be simple objects ("scalars" in GraphQL terminology) or more complex objects that are composed of scalars. According to the GraphQL documentation a scalar has no sub-fields, and all GraphQL implementations are expected to handle, the following scalar types:

- `Int` - which maps to a Java `int`/`Integer` or `long`/`Long`.
- `Float` - which maps to a Java `float`/`Float` or `double`/`Double`.
- `String` - which maps to a Java `char`, `char[]`, `String`, etc.
- `Boolean` - which maps to a Java `boolean`/`Boolean`.
- `ID` - which is a specialized type serialized like a `String`. Usually, ID types are not intended to be human-readable.

In order for an entity class to be defined in the GraphQL schema, it must meet at least one of the following criteria:

- It must be the return type or parameter (annotated with `@Argument`) of a query or mutation method,
- It must be annotated with `@Type`,
- It must be annotated with `@InputType`

POJOs

Any Plain Old Java Object (POJO) can be an entity. No special annotations are required. Implementations of MicroProfile GraphQL must use JSON-B to serialize and deserialize entities to JSON, so it is possible to further define entities using JSON-B annotations.

If the entity cannot be serialized by JSON-B, the implementation must return in an internal server error to the client.

Types vs InputTypes

GraphQL differentiates types from input types. Input types are entities that are sent by the client as arguments to queries or mutations. Types are entities that are sent from the server to the client as return types from queries or mutations.

In many cases the same Java type can be used for input (sent *from* the client) and output (sent *to* the client), but there are cases where an application may need two different Java types to handle input and output.

The `@Type` annotation is used for output entities while the `@InputType` annotation is used for input entities.

Normally these annotations are unnecessary if the type can be serialized and/or deserialized by JSON-B, and if the type is specified in a query or mutation method. These annotations can be used

to specify the name of the type in the GraphQL schema; by default, the entity name in the schema will be the same as the simple class name of the entity type for output types; for input types, the simple class name is used with "Input" appended. Thus, an entity class named `com.mypkg.Tree` would create a GraphQL schema type called "Tree" and an input type called "TreeInput". These annotations can also be used to add a description to the entity in the schema.

Java interfaces as GraphQL entity types

It is possible for entities (types and input types) to be defined as a Java interfaces. In order for JSON-B to deserialize an interface, the interface may need a `JsonbDeserializer` in order to instantiate a concrete type.

GraphQL interfaces

Limitations

Generic types

TODO: info on limitations to generic types (collections only?)

Fields

Fields in GraphQL are similar to fields in Java in that they are a child of a single entity. Thus, Java fields on entity classes are, by default, GraphQL fields of that entity. It is also possible for GraphQL fields that are not part of the Java entity object to be represented as a field of the GraphQL entity. This is because all GraphQL fields are also queries.

Consider the following example:

```
public class SuperHero {
    private String name;
    private String realName;
    private List<String> superPowers;
    // ...
}
```

The Java fields, `name`, `realName` and `superPowers` are all GraphQL fields of the `SuperHero` entity type. Now consider this example:

```
@GraphQLApi
public class MyQueries {

    @Query
    public Location currentLocation(@Source SuperHero hero) {
        return getLocationForHero(hero.getName());
    }
    // ...
}
```

The above query adds a new field to the `SuperHero` GraphQL entity type, called `currentLocation`. This field is not part of the `SuperHero` Java class, but *is* part of the GraphQL entity. This association is made by using the `@Source` annotation. Also note that the `currentLocation` method will only be invoked if the client requests the `currentLocation` field in the query. This is a useful way to prevent looking up data on the server that the client is not interested in.

Deprecation

Application developers can mark entity fields as deprecated in the GraphQL schema with the `@Deprecated` annotation - either the annotation that comes from the JDK in the `java.lang` package or the MicroProfile GraphQL annotation in the `org.eclipse.microprofile.graphql` package. The latter annotation allows the developer to specify additional text that might provide more information to the consumers of the schema.

Default Values

The `@DefaultValue` annotation may be used to specify a value in an input type to be used if the client did not specify a value. Default values may only be specified on input types (and also as `@Argument` parameters) and will have no effect if specified on output types. The value specified in this annotation may be plain text for Java primitives and `String` types or JSON for complex types.

FieldsOrder / InputFieldsOrder

TODO: info on fields ordering

Ignorable fields

There may be cases where a developer wants to use a class as a GraphQL type or input type, but use fields that should not be part of the exposed schema. The `@Ignore` annotation can be placed on the field to prevent it from being part of the schema.

GraphQL Components

Component Definition

API Annotation

GraphQL endpoints must be annotated with the `@GraphQLApi` annotation.

The `@GraphQLApi` annotation is defined in the API:

GraphQLApiAnnotation

```
package org.eclipse.microprofile.graphql;

//...

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GraphQLApi {
}
```

Basic Example

Example

```
@GraphQLApi
@RequestScoped
public class MembershipGraphQLApi {

    @Inject
    private MembershipService membershipService;

    @Query("memberships")
    public List<Membership> getAllMemberships() {
        return getAllMemberships(Optional.empty());
    }

    // Other GraphQL queries and mutations
}
```

Queries

Queries allows a user to ask for all or specific fields on an object.

API Annotation

For classes that are annotated with `@GraphQLApi`, implementations must create a query in the schema for every method that is annotated with `@Query`.

The `@Query` annotation is defined in the API:

QueryAnnotation

```
package org.eclipse.microprofile.graphql;

//...

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Query {
    String value() default "";
    String description() default "";
}
```

Table 1. Parameters

value	Overrides the field name in the resulting schema, default will be the method name
description	Sets a description on the field, default will be null

Basic POJO Example

Example

```
@Query
public SuperHero superHero(@Argument("name") String name) {
    return heroDB.getHero(name);
}
```

Entity fields are also queries

In the previous example, an explicitly defined query named "superHero" returns a `SuperHero` entity. The fields on that entity class are also implicitly defined as queries. It is possible to define fields as queries explicitly by using the `@Source` annotation on a parameter to the query method. More information on this is available in the Entity Fields section.

Unnamed queries

Mutations

TODO: info on how to create mutations

Generated Schema

TODO: info on how the schema should look based on annotations in component classes

Deprecation

TODO: info on how to use the `@Deprecated` annotation appropriately

Default Values

TODO: info on how to use the `@DefaultValue` annotation appropriately

Lifecycle

TODO: info on the components' lifecycle - based on CDI scoping

Error Handling

In GraphQL applications most errors will either be client, server or transport errors.

Client errors occur when the client has submitted an invalid request. Examples of client errors include specifying a query or mutation that does not exist, requesting a field on an entity that does not exist, specifying the wrong type of data (such as specifying an `Int` when the schema requires a `String`), etc.

Server errors occur when the request is valid and is properly transported to the server application but the response is unexpected or unable to be fulfilled. Examples of server errors include bugs in the application code, a back-end resource such as a database is down, etc.

Transport errors occur when the request cannot be delivered to the server or when the response cannot be delivered to the client. Examples of transport errors include network disruption, mis-configured firewalls, etc.

The MP GraphQL specification addresses the handling of client and server errors. Transport error handling is beyond the scope of this document.

Client Errors

Client errors must be handled automatically by the implementation. Invalid requests must never result in user application code invocation. Instead, the implementation must provide the client with an error message that indicates why the client request was invalid.

Server Errors

If the client request is valid, then the implementation must invoke the correct query or mutation method in the user application. The user application can indicate that an error has occurred by throwing an exception (checked or unchecked). When the user application throws and exception, the implementation must send back a response that includes an error message.

The user may determine the error message that is sent back to the client in two ways: - Throw an instance of `GraphQLException` or a subclass. The implementation must send the exception's message text to the client. Optionally, the user can specify an `ExceptionType` in the exception which must also be sent to the client if specified. - Specify the default error message to use when any other (non-`GraphQLException`) exception is thrown. This is set using the MicroProfile Config property, `mp.graphql.defaultErrorMessage`.

Partial Results

It is possible in GraphQL to send back some results even though the overall request may have failed. This is possible by passing the partial results to the `GraphQLException` (or subclass of `GraphQLException`) that is thrown by the query or mutation method. For example:

```
@Query
public Collection<SuperHero> allHeroesFromCalifornia() throws GraphQLException {
    List<SuperHero> westCoastHeroes = new ArrayList<>();
    try {
        for (SuperHero hero : database.getAllHeroes()) {
            if (hero.getPrimaryLocation().contains("California")) {
                westCoastHeroes.add(hero);
            }
        }
    } catch (Exception ex) {
        throw new GraphQLException(ex, westCoastHeroes);
    }
    return westCoastHeroes;
}
```

If an exception is thrown while iterating over of the database collection of heroes or while checking a hero's location, all previously-processed heroes will still be in the list and will be displayed to the client along with the error data.

Note that the `partialResults` object passed to the `GraphQLException` must match the return type of the query/mutation method from which it is thrown. Otherwise the implementation must throw a `ClassCastException` internally resulting in a much less usable result returned to the client.

It is also possible to send partial results when using multiple methods and the `@Source` annotation. Here is an example:

```
@Query
public Collection<SuperHero> allHeroes() {
    return database.getAllHeroes();
}

@Query
public Location currentLocation(@Source SuperHero hero) throws GraphQLException {
    if (hero.hasLocationBlockingPower()) {
        throw new GraphQLException("Unable to determine location for " +
hero.getName());
    }
    return database.getLocationForHero(hero);
}
```

Suppose the client issued this query:

```
query allHeroes {
    allHeroes {
        name
        currentLocation
    }
}
```

In this case, if there are any heroes that have a location blocking power, one or more errors will be returned to the client. However, the names of all of the heroes in the database will be returned as well as the location of all heroes to do not have a location blocking power.

Release Notes for MicroProfile GraphQL 1.0

MicroProfile GraphQL Spec PDF MicroProfile GraphQL Spec HTML MicroProfile GraphQL Spec Javadocs

Key features:

- Code-first approach to GraphQL schema generation.