MicroProfile GraphQL

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

1.0.0-RC1, February 14, 2020: Draft

Table of Contents

Specification: MicroProfile GraphQL

Version: 1.0.0-RC1

Status: Draft

Release: February 14, 2020

MicroProfile GraphQL

Introduction to 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 makes 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.

There are 2 main requirements for all implementations of this specification, namely:

- Generate and make the GraphQL Schema available. This is done by looking at the annotations in the users code, and must include all GraphQL Queries and Mutations as well as all entities as

defined either explicitly by annotations or implicitly as the response type or argument(s) of Queries and Mutations.

- Execute GraphQL requests. This will be in the form of either a Query or a Mutation. As a minimum the specification must support executing these requests via HTTP.

GraphQL Entities

Entities are the objects used in GraphQL. They can be:

- Scalars, or simple objects ("scalars" in GraphQL terminology),
- Enumerable types (similar to Java Enum),
- Complex objects that are composed of scalars and/or enums and/or other complex objects and/or collections of these.

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`.
- `Float` - which maps to a Java `float`/`Float` or `double`/`Double`.
- `String` - which maps to a Java `String`.
- `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.

Note that an ID scalar must map to a Java `String`, numerical primitive `long` and `int` or their object equivalents (`Long`, `Integer`), or a `java.util.UUID` - anything else is considered a deployment error.

GraphQL allows implementations to define additional scalars. MicroProfile GraphQL implementations are required to handle the following additional scalar types:

- `Short` - which maps to a Java `short`/`Short`.
- `Long` - which maps to a Java `long`/`Long`.
- `Char` - which maps to a Java `char`,`Character`.
- `Byte` - which maps to a Java `byte`/`Byte`.
- `BigInteger` - which maps to a Java `BigInteger`.
- `BigDecimal` - which maps to a Java BigDecimal`.
- `Date` - which maps to a Java `java.time.LocalDate`.
- `Time` - which maps to a Java `java.time.LocalTime` or `java.time.OffsetTime`.
- `DateTime` - which maps to a Java `java.time.LocalDateTime`, `java.time.OffsetDateTime` or `java.time.ZonedDateTime`.

Implementations may define additional custom scalars beyond those listed above.

Numbers

All number scalars (Int, Double, Float, Short, Long, Byte, BigInteger and BigDecimal) can be

formatted using the `@NumberFormat` or alternatively the JSON-B annotation `@JsonbNumberFormat` (except that `@JsonbNumberFormat` is not valid on `TYPE_USE`)

In the case that a property has both `@NumberFormat` and `@JsonbNumberFormat`, the GraphQL annotation (`@NumberFormat`) takes priority.

When formatting is added to a number type, the formatted result will be of type String.

Example:

```
@JsonbNumberFormat(value = "¤000",locale = "en-ZA")
private Short formattedShortObject;
```

will result in a formatted amount (South African Rand) in the result:

```
{
    "data": {
        "testScalarsInPojo": {
            "formattedShortObject": "R123"
        }
    }
}
```

List example:

```
@Mutation
public SuperHero trackHeroLongLat(@Name("name") String name,
                                  @Name("coordinates") List<List<@NumberFormat(
"00.0000000 'longlat'") BigDecimal>> coordinates) throws UnknownHeroException {
    // Here add the tracking
    return superHero;
}
```

Dates

By default the date related scalars (DateTime, Date, and Time) will use a ISO format.

- `yyyy-MM-dd'T'HH:mm:ss` for DateTime
- `yyyy-MM-dd'T'HH:mm:ssZ` for OffsetDateTime
- `yyyy-MM-dd'T'HH:mm:ssZ'['VV']'` for ZonedDateTime
- `yyyy-MM-dd` for Date
- `HH:mm:ss` for Time
- `HH:mm:ssZ` for OffsetTime

By adding the `@DateFormat` annotation, or alternatively JSON-B annotation `@JsonbDateFormat`, a user

can change the format. However, `@JsonbDateFormat` does not support usage on `TYPE_USE`.

In the case that a property has both `@DateFormat` and `@JsonbDateFormat`, the GraphQL annotation (`@DateFormat`) takes priority.

Enumerable types

GraphQL offers enumerable types similar to Java `enum` types. In order for an enum to be defined in the GraphQL schema, it must meet the following criteria:

- It must be the return type or parameter of a query or mutation method,

Optionally it can also:

- Be annotated with `@Enum` with a value to name the type
- Be annotated with `@Name` with a value to name the type

The implementation will produce the GraphQL `enum` type in the schema. For example:

```
@Type
public class SuperHero {
    private ShirtSize tshirtSize; // public getters/setters, ...

    @Enum("ClothingSize")
    public enum ShirtSize {
        S, M, L, XL
    }
}
```

The implementation would generate a schema that would include:

```
enum ClothingSize {
  L
  M
  S
  XL
}

type SuperHero {
  #...
  tshirtSize: ClothingSize
  #...
}

input SuperHeroInput
  #...
  tshirtSize: ClothingSize
  #...
}
#...
```

When using an enumerated type, it is considered a validation error when the client enters a value that is not included in the enumerated type.

Complex objects

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

- It must be the return type or parameter of a query or mutation method, or the return type of a method that has a `@Source` annotation as a parameter.
- It implements an interface that is the return type of query or mutation method, or the return type of a method that has a `@Source` annotation as a parameter.

Optionally it can also:

- Be annotated with `@Type` (for return objects) to name the field,
- Be annotated with `@Input` (for input parameter) to name the field,
- Be annotated with `@Interface` (for interfaces) to name the field,
- Be annotated with `@Name` to name the field,

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

JSON-B annotations can be used to help determine schema information as well as data transformation at runtime. In all cases, annotations in the `org.eclipse.microprofile.graphql`

package will trump JSON-B annotations if there is a conflict.

If the entity cannot be serialized, the implementation must return in a server error to the client.

Types vs Input

GraphQL differentiates output types and 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 `@Input` annotation is used for input entities.

Normally these annotations are unnecessary if the type can be serialized and/or deserialized, 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".

GraphQL interfaces

It is possible for output types to be defined as a Java interfaces. GraphQL interfaces are very similar in concept to Java interfaces, in that other types may implement an interface. This allows the GraphQL schema to better align with the Java application's model and allows clients to retrieve the same data (fields) on multiple different entity types. GraphQL interfaces are created with a Java interface type and might be annotated with `@Interface`. The MP GraphQL implementation must then generate a schema where every class in the application that implements that Java interface must have a type in the schema that implements the GraphQL interface. For example:

```
@Interface
public interface Character {
    public String getName();
}

public class SuperHero implements Character {

    private String name;

    @Override
    @Description("Name of hero")
    public String getName() { return name; }

    // ...
}

public class Villain implements Character {

    private String name;

    @Override
    @Description("Name of villain")
    public String getName() { return name; }

    // ...
}
```

This should generate a schema like:

```
interface Character {
  name: String
}

type SuperHero implements Character {
  "Name of hero"
  name: String
  #...
}

type Villain implements Character {
  "Name of villain"
  name: String
  #...
}
```

Limitations

At the moment the spec does not support interfaces on input types.

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 {

    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.

The Java code example above will add a field to the SuperHero type in the schema:

```
type SuperHero {
    #...
    currentLocation: String
    #...
}
```

You can also choose to expose the method containing the `@Source` annotation as a top-level `Query` by adding the `@Query` annotation:

```
@GraphQLApi
public class MyQueries {

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

Above will create the field on SuperHero as described before, and will also add a Query like this:

```
"Query root"
type Query {
    #...
    currentLocation(arg0: SuperHeroInput): String
    #...
}
```

Naming

Users can use the `@Name` annotation (or `@JsonbProperty` from JsonB API) to specify a different field name for the field in the GraphQL schema. In all cases, the `@Name` annotation will take precedence over the `@JsonbProperty` annotation, if they are both specified.

For example:

Name Java Code Example

```
public class Widget {
    @Name("widgetName")
    private String name;
    private double weight;
    private int quantity;
    //...

    @JsonbProperty("shippingWeight")
    public double getWeight() {
        return weight;
    }
    //...

    @Name("qty")
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

That would create a schema like this:

Name Schema Example

```
type Widget {
    widgetName: String
    quantity: Int!
    shippingWeight: Double!
}

input WidgetInput {
    widgetName: String
    qty: Int!
    weight: Double!
}
```

By putting the `@Name` (or `JsonbProperty`) annotation on the `getter` method, rather than the field, the name will only apply to the `Type`, eg:

```
public class Widget {

    private float price;

    @Name("cost")
    public float getPrice(){
        return this.price;
    }

    public void setPrice(float price){
        this.price = price;
    }
}
```

This would result in a schema that looks something like:

```
type Widget {
    cost: Float!
}
input WidgetInput {
    price: Float!
}
```

The input type keeps the default field name. Similarly, when the `@Name` (or `JsonbProperty`) annotation is only placed on the `setter` method, the name will only apply to the `Input`, eg:

```
public class Widget {

    private float price;

    public float getPrice(){
        return this.price;
    }

    @Name("cost")
    public void setPrice(float price){
        this.price = price;
    }
}
```

This would result in a schema that looks something like:

```
type Widget {
    price: Float!
}
input WidgetInput {
    cost: Float!
}
```

When the default name is used, i.e, there is no annotation specifying the name, the field name will always be used, and not the method name.

The same applies to `Query` and `Mutation` methods. If that method starts with `get`, `set` or `is`, that will be removed when determining the name. Eg:

```
@GraphQLApi
public class MyQueries {

    @Query
    public Location getCurrentLocation(@Source SuperHero hero) {
        // ...
    }
}
```

This would result in a schema that looks something like this:

```
"Query root"
type Query {
  #...
  currentLocation(arg0: SuperHeroInput): String
  #...
}
```

Note that the `get` is removed from the name in the schema.

Even though `@Name` is not required on an input argument for a `@Query` or `@Mutation`, it is strongly recommended as it is the only guaranteed portable way to ensure the argument names.

If no name is provided using the `@Name` annotation and the user compiled the application class with the `-parameters` option, then the implementation should use the Java parameter names as the schema argument names.

Example recommended argument usage (with annotation):

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

Above will result in:

```
"Query root"
type Query {
  # ...
  superHero(name: String): SuperHero
  # ...
```

If the `@Name` annotation is not present, and the user did not compile with the `-parameters` option, the generated schema will use generic argument names like `arg0`, `arg1` and so on.

Example argument usage (with no annotation):

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

Above will result in:

```
"Query root"
type Query {
  # ...
  superHero(arg0: String): SuperHero
  # ...
```

When adding a `@Name` to a `@Source` method, you can name the field that should be added to the type, eg:

```
@GraphQLApi
public class MyQueries {

    @Name("heroLocation")
    public Location getCurrentLocation(@Source SuperHero hero) {
        // ...
    }
}
```

Above will result in the schema like this:

```
type SuperHero {
    #...
    heroLocation: String
    #...
}
```

Also making this a `Query` by adding the `@Query` annotation:

```
@GraphQLApi
public class MyQueries {

    @Query
    @Name("heroLocation")
    public Location getCurrentLocation(@Source SuperHero hero) {
        // ...
    }
}
```

will result in:

```
"Query root"
type Query {
    #...
    heroLocation(arg0: SuperHeroInput): String
    #...
}
```

If you want the field name generated in `SuperHero` and the query name to be different, you can name the Query like this:

```
@GraphQLApi
public class MyQueries {

    @Query("locationQuery")
    @Name("heroLocation")
    public Location getCurrentLocation(@Source SuperHero hero) {
        // ...
    }
}
```

will result in:

```
"Query root"
type Query {
    #...
    locationQuery(arg0: SuperHeroInput): String
    #...
}

type SuperHero {
    #...
    heroLocation: String
    #...
}
```

As with any argument, you can also name the argument in the above scenario:

```
@GraphQLApi
public class MyQueries {

    @Query("locationQuery")
    public Location getCurrentLocation(@Name("heroInput") @Source SuperHero hero) {
        // ...
    }
}
```

will result in:

```
"Query root"
type Query {
    #...
    locationQuery(heroInput: SuperHeroInput): String
    #...
}
```

Description

The `@Description` annotation can be used to provide a description in the generated schema for entity types (both input and output types) and fields.

example:

```
@Query
@Description("List all super heroes in the database")
public Collection<SuperHero> allHeroes() {
    // ...
}
```

will result in:

```
"Query root"
type Query {
    "List all super heroes in the database"
    allHeroes: [SuperHero]
    #...
}
```

Default Descriptions

Formatting annotations like `@NumberFormat` and `@DateFormat` (or `@JsonbDateFormat` and `@JsonbNumberFormat` from JsonB) can be used to transform data at runtime (see Scalars)

If no `@Description` annotation is provided for a date or number field, the format string specified in the relative annotation (`@NumberFormat` or `@DateFormat` or `@JsonbDateFormat` or `@JsonbNumberFormat`) will be used as the field's description, or, for dates only, the default date format in the case there is no `@DateFormat` or `@JsonbDateFormat` annotation.

If a `@Description` annotation is provided for a date or number field, the format string specified in the relative annotation (`@NumberFormat` or `@DateFormat` or `@JsonbDateFormat` or `@JsonbNumberFormat`) will be appended to the given description in brackets `(⋯)`, or, for dates only, the default date format in the case there is no `@DateFormat` or `@JsonbDateFormat` annotation.

Example:

```
private LocalDate dateObject;

@Description("This is another date")
private LocalDate anotherDateObject;

@JsonbDateFormat("MM dd yyyy")
@Description("This is a formatted date")
private LocalDate formattedDateObject;

@JsonbNumberFormat("#0.0")
private Float formattedFloatObject;

@Description("This is a formatted number")
@JsonbNumberFormat("#0.0")
private Double formattedDoubleObject;
```

will result in:

```
type ScalarHolder {
  "yyyy-MM-dd"
  dateObject: Date

  "This is another date (yyyy-MM-dd)"
  anotherDateObject: Date

  "This is a formatted date (MM dd yyyy)"
  formattedDateObject: Date

  "#0.0"
  formattedFloatObject: Float

  "This is a formatted number (#0.0)"
  formattedDoubleObject: Double

  #...
}
```

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 method parameters (including method parameter for the query/mutation) 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.

example:

DefaultValue Example

```
@Query
public Collection<SuperHero> allHeroesIn(
    @DefaultValue("New York, NY") @Name("city") String city) {

    return allHeroesByFilter(hero -> {
        return city.equals(hero.getPrimaryLocation());});
    }
}

public final static String CAPE =
        "{" +
        "   \"id\": 1000," +
        "   \"name\": \"Cape\","+
        "   \"powerLevel\": 3," +
        "   \"height\": 1.2," +
        "   \"weight\": 0.3," +
        "   \"supernatural\": false" +
        "}";

@Mutation
public SuperHero provisionHero(@Name("hero") String heroName,
                               @DefaultValue(CAPE) @Name("item") Item item)
                               throws UnknownHeroException {

        SuperHero hero = heroDB.getHero(heroName);
        if (hero == null) {
            throw new UnknownHeroException(heroName);
        }
        hero.getEquipment().add(item);
        return hero;
    }
```

The `@DefaultValue` annotation may also be placed on fields and setters on entity classes to specify the default for GraphQL fields.

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.

If the `@Ignore` annotation is placed on the field itself, then the field will be excluded from both the input and output types in the generated schema. If the annotation is only placed on the "getter" method, then it will only be excluded from the output type. If the annotation is only placed on the "setter" method, then it will only be excluded from the input type.

Similarly, the `@JsonbTransient` annotation (from Json-B API) can be used to ignore certain fields from the type or input type in the schema. The same rules apply: if the annotation is on the getter,

then the field is ignored in the type; if the annotation is on the setter, then the field is ignored in the input type; if the annotation is on the Java field, it is ignored in both.

example:

Ignore Java Code Example

```
public class Widget {
    @Ignore
    private String name;
    private double weight;
    private int quantity;
    //...

    @JsonbTransient
    public double getWeight() {
        return weight;
    }
    //...

    @Ignore
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

That would create a schema like this:

Ignore Schema Example

```
type Widget {
    quantity: Int!
}

input WidgetInput {
    weight: Double!
}
```

Non-nullable fields

The GraphQL specification states that fields may be marked as non-nullable - the field's type is marked with an exclamation point to indicate that null values are not allowed. Non-nullable fields may be present on types and input types, providing the client with the proper expectations for providing an input type and that they can expect a non-null value on the return type. If the client sends a null value for a required (non-nullable) field or sends an entity with the required (non-nullable) field unspecified, the implementation should respond with a validation error. Likewise, the implementation should return an error if a null is returned for a required (non-nullable) field from the application code.

By default all GraphQL fields generated from Java primitive properties (`boolean`, `int`, `double`, etc.)

will automatically be marked as required. If a Java primitive property has a `@DefaultValue` annotation value, then null is allowed, but the implementation is expected to convert the value to be the default value specified in the annotation.

By default, all GraphQL fields generated from non-primitive properties will be considered nullable. A user may specify that a field is required/non-nullable by adding the `@NonNull` annotation. This annotation may be applied to an entity's getter method, setter method or field. The placement will determine whether it applies to the type, input type or both, respectively.

Example, placing the annotation on the field:

```
public class Item {
    // ...
    @NonNull
    private String name;
    // ....
```

Will result in both the Type and Input to be marked not null (!):

```
input ItemInput {
  name: String!
  #...
}

type Item {
  name: String!
  #...
}
```

The annotation can also be use to indicate that elements in a collection can not be null, example:

```
public class SuperHero {
    // ...
    private List<@NonNull String> superPowers;
    // ...
}
```

This indicates that superPowers can be null, but if it's not, then it must only contain non-null entries.

The code above will result in a schema entry like this:

```
type SuperHero {
  superPowers: [String!]
  #...
}

input SuperHeroInput {
  superPowers: [String!]
  #...
}
```

Placing a `@NonNull` on the List can also makes the actual list non null.

The implementation should ignore a `@NonNull` annotation when it is on the same field or setter method that also contains `@DefaultValue` annotation, as the "null" value would result in the default value being used.

One drawback to using non-nullable fields is that if there is an error loading a child field, that error could propagate itself up causing the field to be null - and since this is itself an error condition, the implementation must return the non-null field error, which means that the implementation would not be able to send partial results for other child fields.

GraphQL Components

Component Definition

API Annotation

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

The `@GraphQLApi` annotation is a marker annotation that identifies a CDI Bean as a GraphQL Endpoint

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 or collection of objects.

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's name can be specified in the value parameter of the `@Query` annotation, or is generated from the method name if no annotation value is provided.

Basic POJO Example

Example

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

Note that generic types other than subtypes of `java.util.Collection` (such as `java.util.List` or `java.util.Set`) are not allowed to be specified as query return types. Implementations may allow additional types (such as `java.util.Map`), but the behavior for these return types are undefined.

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 Fields section.

Name

The name of a query in the schema is obtained using the following order:

- if the method is annotated with a `@Query` annotation containing a non-empty String for it's value, that String value is used as the query name.
- if the method is annotated with a `@Name` annotation containing a non-empty String for it's value, that String value is used as the query name.
- if the method is annotated with a `@JsonbProperty` annotation containing a non-empty String for it's value, that String value is used as the query name.
- if no other name can be determined, the Java method name is used as the query name. (with the get/is removed if this is a getter)

Note that it is considered a deployment error if more than one query method has the same name with the same arguments.

Description

Queries may be documented with descriptions in the schema by adding a `@Description` annotation with documentation text as the annotation value to the query method. For example:

DescriptionExample

```
@Query
@Description("Returns the super hero with the specified name")
public SuperHero superHero(@Name("name") String name) {
    return heroDB.getHero(name);
}
```

This would generate a schema that would include:

DescriptionSchemaExample

```
type Query {
  ...
  "Returns the super hero with the specified name"
  superHero(name: String): SuperHero
  #...
----
```

The `@Description` annotation can also be placed on parameters of a query method to provide documentation for the arguments. For example:

ArgumentDescriptionExample

```
@Query
@Description("Returns the super hero with the specified name")
public SuperHero superHero(@Name("name") @Description("Super hero name, not real name
") String name) {
    return heroDB.getHero(name);
}
```

This would generate a schema that would include:

ArgumentDescriptionSchemaExample

```
type Query {
  ...
  "Returns the super hero with the specified name"
  superHero(
  "Super hero name, not real name"
  name: String
  ): SuperHero
  #...
----
```

Void Queries

By it's very nature, query methods must return some value, thus it is considered a deployment error for a method with a `void` return type to be annotated with `@Query`. If a void method is annotated with the `@Query` annotation, the implementation must prevent the application from starting and should provide a log message indicating that this is not allowed.

Mutations

While queries are intended for clients to read data, mutations are intended to modify data. Mutations may create new entities or update or delete existing entities.

API Annotation

Like queries, mutation methods must be in a class annotated with the `@GraphQLApi` annotation.

Mutation methods are annotated with `@Mutation`. The name of the mutation is specified in the value of the annotation or is generated from the method name if no annotation value is provided.

Mutations generally require arguments (parameters) in order to determine which entity to modify/delete or the data necessary to create a new entity, etc. These arguments are the parameters of the mutation method and should (not mandatory) be annotated with `@Name`. The `@Name` annotation's value is used to specify the name of the argument. The argument name will be used in the generated schema. Arguments can be GraphQL scalars, enums or more complex input types (for more information on input types, see Types vs Input).

Basic POJO Examples

Create Example

```
@Mutation
public SuperHero createNewHero(@Name("hero") SuperHero newHero)
    throws DuplicateSuperHeroException {

    heroDB.addHero(newHero);
    return heroDB.getHero(newHero.getName());
}
```

Delete Example

```
@Mutation
public SuperHero removeHero(@Name("hero") String heroName)
    throws UnknownHeroException {

    SuperHero removedHero = heroDB.removeHero(heroName);
    if (removedHero == null) {
        throw new UnknownHeroException(heroName);
    }
    return removedHero;
    }
```

Update Example

```
@Mutation
public SuperHero addNewPowerToHero(@Name("hero") SuperHero hero,
                                   @Name("newPower") String newPower)
                                   throws UnknownHeroException {

    SuperHero heroFromDB = heroDB.getHero(newHero.getName());
    if (heroFromDB == null) {
        throw new UnknownHeroException(hero.getName());
    }
    heroFromDB.getSuperPowers().add(newPower);
    return heroFromDB;
}
```

Note that generic types other than subtypes of `java.util.Collection` (such as `java.util.List` or `java.util.Set`) are not allowed to be specified as mutation return types. Implementations may allow additional types (such as `java.util.Map`), but the behavior for these return types are undefined.

Names

The name of a mutation in the schema is obtained using the following order:

- if the method is annotated with a `@Mutation` annotation containing a non-empty String for it's value, that String value is used as the mutation name.
- if the method is annotated with a `@Name` annotation containing a non-empty String for it's value, that String value is used as the mutation name.
- if the method is annotated with a `@JsonbProperty` annotation containing a non-empty String for it's value, that String value is used as the mutation name.
- if no other name can be determined, the Java method name is used as the mutation name. (with the set removed if this is a setter)

Note that it is considered a deployment error if more than one mutation method has the same name with the same arguments.

Description

As with Queries, Mutations may be documented with descriptions in the schema by adding a `@Description` annotation with documentation text as the annotation value to the mutation method. See the example in the Query section for more details.

Void Mutations

Like query methods, mutation methods are expected to return some value, thus it is considered a deployment error for a method with a `void` return type to be annotated with `@Mutation`. If a void method is annotated with the `@Mutation` annotation, the implementation must prevent the application from starting and should provide a log message indicating that this is not allowed.

Generated Schema

MicroProfile GraphQL uses a "code first" approach so that developers do not need to manually keep the code and schema in sync. Each MP GraphQL application will still have a schema but it is generated by the MP GraphQL implementation.

The schema must be available at the `/graphql/schema.graphql` location, relative to the context root.

For example, suppose your application was registered at host, "myhost" on TCP port "50080" with a context root of "MyApp" (usually the context root is the name of the WAR file but without the file extension), then the schema would be available at: `http://myhost:50080/MyApp/graphql/schema.graphql`

Arguments

Arguments can exist for both queries and mutations and are generally represented as method parameters in Java code.

For example:

Basic Argument Example

```
@Query
public List<SuperHero> getHeroesAt(@Name("location") String location) { //...
```

In this example, `location` is an argument that the client must provide when invoking the `getHeroesAt` query.

Note that abstract classes, interfaces or generic types other than subtypes of `java.util.Collection` (such as `java.util.List` or `java.util.Set`) are not allowed to be specified as arguments. Implementations may allow additional types (such as `java.util.Map`), but the behavior for these types of arguments are undefined.

Default Values

It is possible to specify a default value for query or mutation arguments so that the client does not need to specify a value. This is done via the `@DefaultValue` annotation. Read more about this in the Default Values section.

Lifecycle

MicroProfile GraphQL components (POJOs annotated with `@GraphQLApi`) are CDI beans. As such, their lifecycle is managed by CDI. Request-scoped components should be constructed per-request, and application-scoped components should exist for the lifetime of the application. One exception to the normal scoping is `@Dependent` - this scope is treated as if it were a singleton.

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.

Structure

As per the GraphQL Specification, errors can be included in the response, and each error can contain the following:

- `message` - a string description of the error (mandatory)
- `locations` - a list of locations, where each location is a map with the keys `line` and `column`, both positive numbers starting from 1 which describe the beginning of an associated syntax element to a particular point in the requested GraphQL document. (optional)
- `path` - details the path of the response field which experienced the error (optional)
- `extensions` - this allows implementation to add any additional key-value pairs of information (optional)

Error Example

```
{
    "errors": [
        {
            "message": "Validation error of type WrongType: argument 'powerLevel' with
value 'StringValue{value='Unlimited'}' is not a valid 'Int' @ 'updateItemPowerLevel'",
            "locations": [
                {
                    "line": 2,
                    "column": 37
                }
            ],
            "extensions": {
                "description": "argument 'powerLevel' with value
'StringValue{value='Unlimited'}' is not a valid 'Int'",
                "validationErrorType": "WrongType",
                "queryPath": [
                    "updateItemPowerLevel"
                ],
                "classification": "ValidationError"
            }
        }
    ],
    "data": null
}
```

In the example above you can see implementation specific usage of the `extensions` section to add more information on the validation error.

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.

Unchecked exceptions

If an unchecked exception is thrown from the user application, the implementation must 'hide' the error message (for security reasons) and replace the `message` with a configured default message.

The default message is "Server Error" and can be configured by the user using MicroProfile Config

and setting the `mp.graphql.defaultErrorMessage` property.

Example:

```
mp.graphql.defaultErrorMessage=Unexpected failure in the system. Jarvis is working to fix it.
```

Users can allow unchecked exception messages to be included (changing the default behavior as described above) by adding the exception class name to a `whitelist`. This is done using MicroProfile Config and setting the `mp.graphql.exceptionsWhiteList` property (comma-separated list)

Example:

```
mp.graphql.exceptionsWhiteList=org.eclipse.microprofile.graphql.tck.apps.superhero.api.WeaknessNotFoundException
```

(Note: By default all checked exceptions is on the `whitelist` and all unchecked exception on the `blacklist`)

Checked exceptions

By default checked exceptions must include the exception message in the `message` field of the error, and where possible also include the `locations` and `path`.

Implementations must support the ability to 'hide' this message for checked exceptions by allowing users to add them to a `blacklist`. This is done using MicroProfile Config and setting the `mp.graphql.exceptionsBlackList` property (comma-separated list)

Example:

```
mp.graphql.exceptionsBlackList=java.io.IOException,java.util.concurrent.TimeoutException
```

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 that 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.