graphql-java Documentation

Release stable

Jul 03, 2017

Documentation

This is a GraphQL Java implementation based on the specification and the JavaScript reference implementation.

Status: Version `3.0.0` is released.

The versioning follows Semantic Versioning since `2.0.0`.

Make sure to check out the awesome related projects built on `graphql-java`.

CHAPTER 1

Code of Conduct

Please note that this project is released with a Contributor Code of Conduct. By contributing to this project (commenting or opening PR/Issues etc) you are agreeing to follow this conduct, so please take the time to read it.

CHAPTER 2

Questions and discussions

If you have a question or want to discuss anything else related to this project:

- Feel free to open a new Issue
- We have a public chat room (Gitter.im) for graphql-java: https://gitter.im/graphql-java/graphql-java

CHAPTER 3

License

graphql-java is licensed under the MIT License.

Getting started

`graphql-java` is requires at least Java 8.

How to use the latest release with Gradle

Make sure `mavenCentral` is among your repos:

```
repositories {
    mavenCentral()
}
```

Dependency:

```
dependencies {
  compile 'com.graphql-java:graphql-java:2.4.0'
}
```

How to use the latest release with Maven

Dependency:

```
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java</artifactId>
    <version>2.4.0</version>
</dependency>
```

Hello World

This is the famous "hello world" in `graphql-java`:

```
import graphql.GraphQL;
import graphql.schema.GraphQLObjectType;
import graphql.schema.GraphQLSchema;

import java.util.Map;

import static graphql.Scalars.GraphQLString;
import static graphql.schema.GraphQLFieldDefinition.newFieldDefinition;
import static graphql.schema.GraphQLObjectType.newObject;

public class HelloWorld {

    public static void main(String[] args) {
        GraphQLObjectType queryType = newObject()
                .name("helloWorldQuery")
                .field(newFieldDefinition()
                        .type(GraphQLString)
                        .name("hello")
                        .staticValue("world"))
                .build();

        GraphQLSchema schema = GraphQLSchema.newSchema()
                .query(queryType)
                .build();

        GraphQL graphQL = GraphQL.newGraphQL(schema).build();

        Map<String, Object> result = graphQL.execute("{hello}").getData();
        System.out.println(result);
        // Prints: {hello=world}
    }
}
```

Using the latest development build

The latest development build is available on Bintray.

Please look at Latest Build for the latest version value.

How to use the latest build with Gradle

Add the repositories:

```
repositories {
    mavenCentral()
    maven { url  "http://dl.bintray.com/andimarek/graphql-java" }
}
```

Dependency:

```
dependencies {
  compile 'com.graphql-java:graphql-java:INSERT_LATEST_VERSION_HERE'
}
```

How to use the latest build with Maven

Add the repository:

```
<repository>
    <snapshots>
        <enabled>false</enabled>
    </snapshots>
    <id>bintray-andimarek-graphql-java</id>
    <name>bintray</name>
    <url>http://dl.bintray.com/andimarek/graphql-java</url>
</repository>
```

Dependency:

```
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java</artifactId>
    <version>INSERT_LATEST_VERSION_HERE</version>
</dependency>
```

Creating a schema

A schema defines your GraphQL API by defining each field that can be queried or mutated.

`graphql-java` offers two different ways of defining the schema: Programmatically as Java code or via a special graphql dsl (called IDL).

NOTE: IDL is not currently part of the formal graphql spec. The implementation in this library is based off the reference implementation. However plenty of code out there is based on this IDL syntax and hence you can be fairly confident that you are building on solid technology ground.

If you are unsure which option to use we recommend the IDL.

IDL example:

```
type Foo {
    bar: String
}
```

Java code example:

```
GraphQLObjectType fooType = newObject()
    .name("Foo")
    .field(newFieldDefinition()
            .name("bar")
            .type(GraphQLString))
    .build();
```

DataFetcher and TypeResolver

A `DataFetcher` provides the data for a field (and changes something, if it is a mutation).

Every field definition has a `DataFetcher`. When no one is configured, a PropertyDataFetcher is used.

`PropertyDataFetcher` fetches data from `Map` and Java Beans. So when the field name matches the Map key or the property name of the source Object, no `DataFetcher` is needed.

A `TypeResolver` helps `graphql-java` to decide which type a concrete value belongs to. This is needed for `Interface` and `Union`.

IDL

When defining a schema via IDL, you provide the needed `DataFetcher` and `TypeResolver` when the schema is created:

Example:

```
schema {
    query: QueryType
}

type QueryType {
    hero(episode: Episode): Character
    human(id : String) : Human
    droid(id: ID!): Droid
}


enum Episode {
    NEWHOPE
    EMPIRE
    JEDI
}

interface Character {
    id: ID!
    name: String!
    friends: [Character]
    appearsIn: [Episode]!
}

type Human implements Character {
    id: ID!
    name: String!
    friends: [Character]
    appearsIn: [Episode]!
    homePlanet: String
}

type Droid implements Character {
    id: ID!
    name: String!
    friends: [Character]
    appearsIn: [Episode]!
    primaryFunction: String
}
```

You could generate an executable schema via

```
SchemaParser schemaParser = new SchemaParser();
SchemaGenerator schemaGenerator = new SchemaGenerator();

File schemaFile = loadSchema("starWarsSchema.graphqls");

TypeDefinitionRegistry typeRegistry = schemaParser.parse(schemaFile);
RuntimeWiring wiring = buildRuntimeWiring();
GraphQLSchema graphQLSchema = schemaGenerator.makeExecutableSchema(typeRegistry,␣
↪wiring);
```

The static schema definition file has the field and type definitions but you need a runtime wiring to make it a truly executable schema.

The runtime wiring contains ‘‘DataFetcher‘‘s, ‘‘TypeResolvers‘‘s and custom ‘‘Scalar‘‘s that are needed to make a fully executable schema.

You wire this together using this builder pattern

```
RuntimeWiring buildRuntimeWiring() {
    return RuntimeWiring.newRuntimeWiring()
            .scalar(CustomScalar)
            // this uses builder function lambda syntax
            .type("QueryType", typeWiring -> typeWiring
                    .dataFetcher("hero", new StaticDataFetcher(StarWarsData.
↪getArtoo()))
                    .dataFetcher("human", StarWarsData.getHumanDataFetcher())
                    .dataFetcher("droid", StarWarsData.getDroidDataFetcher())
            )
            .type("Human", typeWiring -> typeWiring
                    .dataFetcher("friends", StarWarsData.getFriendsDataFetcher())
            )
            // you can use builder syntax if you don't like the lambda syntax
            .type("Droid", typeWiring -> typeWiring
                    .dataFetcher("friends", StarWarsData.getFriendsDataFetcher())
            )
            // or full builder syntax if that takes your fancy
            .type(
                    newTypeWiring("Character")
                            .typeResolver(StarWarsData.getCharacterTypeResolver())
                            .build()
            )
            .build();
}
```

There is a another way to wiring in type resolvers and data fetchers and that is via the `WiringFactory` interface. This allow for a more dynamic runtime wiring since the IDL definitions can be examined in order to decide what to wire in. You could for example look at IDL directives to help you decide what runtime to create or some other aspect of the IDL definition.

```
RuntimeWiring buildDynamicRuntimeWiring() {
    WiringFactory dynamicWiringFactory = new WiringFactory() {
        @Override
        public boolean providesTypeResolver(TypeDefinitionRegistry registry,␣
↪InterfaceTypeDefinition definition) {
            return getDirective(definition,"specialMarker") != null;
        }
```

```
        @Override
        public boolean providesTypeResolver(TypeDefinitionRegistry registry,␣
↪UnionTypeDefinition definition) {
            return getDirective(definition,"specialMarker") != null;
        }

        @Override
        public TypeResolver getTypeResolver(TypeDefinitionRegistry registry,␣
↪InterfaceTypeDefinition definition) {
            Directive directive  = getDirective(definition,"specialMarker");
            return createTypeResolver(definition,directive);
        }

        @Override
        public TypeResolver getTypeResolver(TypeDefinitionRegistry registry,␣
↪UnionTypeDefinition definition) {
            Directive directive  = getDirective(definition,"specialMarker");
            return createTypeResolver(definition,directive);
        }

        @Override
        public boolean providesDataFetcher(TypeDefinitionRegistry registry,␣
↪FieldDefinition definition) {
            return getDirective(definition,"dataFetcher") != null;
        }

        @Override
        public DataFetcher getDataFetcher(TypeDefinitionRegistry registry,␣
↪FieldDefinition definition) {
            Directive directive = getDirective(definition, "dataFetcher");
            return createDataFetcher(definition,directive);
        }
    };
    return RuntimeWiring.newRuntimeWiring()
            .wiringFactory(dynamicWiringFactory).build();
}
```

Programmatically

When the schema is created programmatically you provide the `DataFetcher` and `TypeResolver` when the type is created:

Example:

```
DataFetcher<Foo> fooDataFetcher = environment -> {
        // environment.getSource() is the value of the surrounding
        // object. In this case described by objectType
        Foo value = perhapsFromDatabase(); // Perhaps getting from a DB or whatever
        return value;
}

GraphQLObjectType objectType = newObject()
        .name("ObjectType")
        .field(newFieldDefinition()
                .name("foo")
                .type(GraphQLString)
```

```
                .dataFetcher(fooDataFetcher))
        .build();
```

Types

The GraphQL type system supports the following kind of types:

- Scalar
- Object
- Interface
- Union
- InputObject
- Enum

Scalar

`graphql-java` supports the following Scalars:

- `GraphQLString`
- `GraphQLBoolean`
- `GraphQLInt`
- `GraphQLFloat`
- `GraphQLID`
- `GraphQLLong`
- `GraphQLShort`
- `GraphQLByte`
- `GraphQLFloat`
- `GraphQLBigDecimal`
- `GraphQLBigInteger`

Object

IDL Example:

```
type SimpsonCharacter {
    name: String
    mainCharacter: Boolean
}
```

Java Example:

```
GraphQLObjectType simpsonCharacter = newObject()
.name("SimpsonCharacter")
.description("A Simpson character")
.field(newFieldDefinition()
```

```
        .name("name")
        .description("The name of the character.")
        .type(GraphQLString))
.field(newFieldDefinition()
        .name("mainCharacter")
        .description("One of the main Simpson characters?")
        .type(GraphQLBoolean))
.build();
```

Interface

IDL Example:

```
interface ComicCharacter {
    name: String;
}
```

Java Example:

```
GraphQLInterfaceType comicCharacter = newInterface()
    .name("ComicCharacter")
    .description("A abstract comic character.")
    .field(newFieldDefinition()
            .name("name")
            .description("The name of the character.")
            .type(GraphQLString))
    .build();
```

Union

IDL Example:

```
interface ComicCharacter {
    name: String;
}
```

Java Example:

```
GraphQLUnionType PetType = newUnionType()
    .name("Pet")
    .possibleType(CatType)
    .possibleType(DogType)
    .typeResolver(new TypeResolver() {
        @Override
        public GraphQLObjectType getType(TypeResolutionEnvironment env) {
            if (env.getObject() instanceof Cat) {
                return CatType;
            }
            if (env.getObject() instanceof Dog) {
                return DogType;
            }
            return null;
        }
    })
    .build();
```

Enum

IDL Example:

```
enum Color {
    RED
    GREEN
    BLUE
}
```

Java Example:

```
GraphQLEnumType colorEnum = newEnum()
    .name("Color")
    .description("Supported colors.")
    .value("RED")
    .value("GREEN")
    .value("BLUE")
    .build();
```

ObjectInputType

IDL Example:

```
input Character {
    name: String
}
```

Java Example:

```
GraphQLInputObjectType inputObjectType = newInputObject()
    .name("inputObjectType")
    .field(newInputObjectField()
            .name("field")
            .type(GraphQLString))
    .build();
```

Type References (recursive types)

GraphQL supports recursive types: For example a `Person` can contain a list of friends of the same type.

To be able to declare such a type, `graphql-java` has a `GraphQLTypeReference` class.

When the schema is created, the `GraphQLTypeReference` is replaced with the actual real type Object.

For example:

```
GraphQLObjectType person = newObject()
    .name("Person")
    .field(newFieldDefinition()
            .name("friends")
            .type(new GraphQLList(new GraphQLTypeReference("Person"))))
    .build();
```

When the schema is declared via IDL, no special handling of recursive types is needed.

Modularising the Schema IDL

Having one one large schema file is not always viable. You can modularise you schema using two techniques.

The first technique is to merge multiple Schema IDL files into one logic unit. In the case below the schema as been split into multiple files and merged all together just before schema generation.

```
SchemaParser schemaParser = new SchemaCompiler();
SchemaGenerator schemaGenerator = new SchemaGenerator();

File schemaFile1 = loadSchema("starWarsSchemaPart1.graphqls");
File schemaFile2 = loadSchema("starWarsSchemaPart2.graphqls");
File schemaFile3 = loadSchema("starWarsSchemaPart3.graphqls");

TypeDefinitionRegistry typeRegistry = new TypeDefinitionRegistry();

// each registry is merged into the main registry
typeRegistry.merge(schemaParser.compile(schemaFile1));
typeRegistry.merge(schemaParser.compile(schemaFile2));
typeRegistry.merge(schemaParser.compile(schemaFile3));

GraphQLSchema graphQLSchema = schemaGenerator.makeExecutableSchema(typeRegistry,␣
↪buildRuntimeWiring());
```

The Graphql IDL type system has another construct for modularising your schema. You can use *type extensions* to add extra fields and interfaces to a type.

Imagine you start with a type like this in one schema file.

```
type Human {
    id: ID!
    name: String!
}
```

Another part of your system can extend this type to add more shape to it.

```
extend type Human implements Character {
    id: ID!
    name: String!
    friends: [Character]
    appearsIn: [Episode]!
}
```

You can have as many extensions as you think sensible. They will be combined in the order in which they are encountered. Duplicate fields will be merged as one (however field re-definitions into new types are not allowed).

```
extend type Human {
    homePlanet: String
}
```

With all these type extensions in place the *Human* type now looks like this at runtime.

```
type Human implements Character {
    id: ID!
    name: String!
    friends: [Character]
    appearsIn: [Episode]!
    homePlanet: String
}
```

This is especially useful at the top level. You can use extension types to add new fields to the top level schema "query". Teams could contribute "sections" on what is being offered as the total graphql query.

```
schema {
  query: CombinedQueryFromMultipleTeams
}

type CombinedQueryFromMultipleTeams {
    createdTimestamp : String
}

# maybe the invoicing system team puts in this set of attributes
extend type CombinedQueryFromMultipleTeams {
    invoicing : Invoicing
}

# and the billing system team puts in this set of attributes
extend type CombinedQueryFromMultipleTeams {
    billing : Billing
}

# and so and so forth
extend type CombinedQueryFromMultipleTeams {
    auditing : Auditing
}
```

Subscription Support

Subscriptions are not officially specified yet: `graphql-java` supports currently a very basic implementation where you can define a subscription in the schema with `GraphQLSchema.Builder.subscription(...)`. This enables you to handle a subscription request:

```
subscription foo {
    # normal graphql query
}
```

Execution

To execute a Query/Mutation against a Schema build a new `GraphQL` Object with the appropriate arguments and then call `execute()`.

The result of a Query is a `ExecutionResult` Object with the result and/or a list of Errors.

Example: [GraphQL Test](src/test/groovy/graphql/GraphQLTest.groovy)

More complex examples: [StarWars query tests](src/test/groovy/graphql/StarWarsQueryTest.groovy)

Mutations

A good starting point to learn more about mutating data in graphql is http://graphql.org/learn/queries/#mutations.

In essence you need to define a `GraphQLObjectType` that takes arguments as input. Those arguments are what you can use to mutate your data store via the data fetcher invoked.

The mutation is invoked via a query like :

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

You need to send in arguments during that mutation operation, in this case for the variables for *$ep* and *$review*

You would create types like this to handle this mutation :

Notice that the input arguments are of type `GraphQLInputObjectType`. This is important. Input arguments can ONLY be of that type and you cannot use output types such as `GraphQLObjectType`. Scalars types are consider both input and output types.

The data fetcher here is responsible for executing the mutation and returning some sensible output values.

```
private DataFetcher mutationDataFetcher() {
    return new DataFetcher() {
        @Override
        public Review get(DataFetchingEnvironment environment) {
            Episode episode = environment.getArgument("episode");
            ReviewInput review = environment.getArgument("review");

            // make a call to your store to mutate your database
            Review updatedReview = reviewStore().update(episode, review);

            // this returns a new view of the data
            return updatedReview;
        }
    };
}
```

Notice how it calls a data store to mutate the backing database and then returns a `Review` object that can be used as the output values to the caller.

Execution strategies

All fields in a SelectionSet are executed serially per default.

You can however provide your own execution strategies, one to use while querying data and one to use when mutating data.

```
ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
        2, /* core pool size 2 thread */
        2, /* max pool size 2 thread */
        30, TimeUnit.SECONDS,
        new LinkedBlockingQueue<Runnable>(),
        new ThreadPoolExecutor.CallerRunsPolicy());

GraphQL graphQL = GraphQL.newObject(StarWarsSchema.starWarsSchema)
        .queryExecutionStrategy(new␣
↪ExecutorServiceExecutionStrategy(threadPoolExecutor))
        .mutationExecutionStrategy(new SimpleExecutionStrategy())
        .build();
```

When provided fields will be executed parallel, except the first level of a mutation operation.

See specification for details.

Alternatively, schemas with nested lists may benefit from using a BatchedExecutionStrategy and creating batched DataFetchers with get() methods annotated @Batched.

Logging

Logging is done with SLF4J. Please have a look at the Manual for details. The `graphql-java` root Logger name is `graphql`.

Contributions

Every contribution to make this project better is welcome: Thank you!

In order to make this a pleasant as possible for everybody involved, here are some tips:

- Respect the [Code of Conduct](#code-of-conduct)
- Before opening an Issue to report a bug, please try the latest development version. It can happen that the problem is already solved.
- Please use Markdown to format your comments properly. If you are not familiar with that: Getting started with writing and formatting on GitHub
- For Pull Requests: * Here are some general tips
  - Please be a as focused and clear as possible and don't mix concerns. This includes refactorings mixed with bug-fixes/features, see [Open Source Contribution Etiquette](http://tirania.org/blog/archive/2010/Dec-31.html)
  - It would be good to add a automatic test. All tests are written in Spock.

Build and test locally

Just clone the repo and type

```
./gradlew build
```

In `build/libs` you will find the jar file.

Running the tests:

```
./gradlew test
```

Installing in the local Maven repository:

```
./gradlew install
```