graphql-java Documentation

Release current

Brad Baker

Nov 08, 2017

Documentation

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

Status: Version `5.0.0` is released.

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.

3.1 Getting started

`graphql-java` requires at least Java 8.

3.1.1 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:3.0.0'
}
```

3.1.2 How to use the latest release with Maven

Dependency:

```
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java</artifactId>
    <version>3.0.0</version>
</dependency>
```

3.1.3 Hello World

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

```
import graphql.ExecutionResult;
import graphql.GraphQL;
import graphql.schema.GraphQLSchema;
import graphql.schema.StaticDataFetcher;
import graphql.schema.idl.RuntimeWiring;
import graphql.schema.idl.SchemaGenerator;
import graphql.schema.idl.SchemaParser;
import graphql.schema.idl.TypeDefinitionRegistry;

import static graphql.schema.idl.RuntimeWiring.newRuntimeWiring;

public class HelloWorld {

    public static void main(String[] args) {
        String schema = "type Query{hello: String}";

        SchemaParser schemaParser = new SchemaParser();
        TypeDefinitionRegistry typeDefinitionRegistry = schemaParser.parse(schema);

        RuntimeWiring runtimeWiring = newRuntimeWiring()
                .type("Query", builder -> builder.dataFetcher("hello", new␣
↪StaticDataFetcher("world")))
                .build();

        SchemaGenerator schemaGenerator = new SchemaGenerator();
        GraphQLSchema graphQLSchema = schemaGenerator.
↪makeExecutableSchema(typeDefinitionRegistry, runtimeWiring);

        GraphQL build = GraphQL.newGraphQL(graphQLSchema).build();
        ExecutionResult executionResult = build.execute("{hello}");

        System.out.println(executionResult.getData().toString());
        // Prints: {hello=world}
    }
}
```

3.1.4 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>
```

3.2 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();
```

3.2.1 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 one is not 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`.

For example imagine you have a `Interface` called *MagicUserType* and it resolves back to a series of Java classes called perhaps *Wizard*, *Witch* and *Necromancer*. The type resolver is responsible for examining a runtime object and deciding what `GraphqlObjectType` should be used to represent it and hence what data fetchers and fields will be invoked.

```
new TypeResolver() {
    @Override
    public GraphQLObjectType getType(TypeResolutionEnvironment env) {
        Object javaObject = env.getObject();
        if (javaObject instanceof Wizard) {
            return (GraphQLObjectType) env.getSchema().getType("WizardType");
        } else if (javaObject instanceof Witch) {
            return (GraphQLObjectType) env.getSchema().getType("WitchType");
        } else {
            return (GraphQLObjectType) env.getSchema().getType("NecromancerType");
        }
    }
};
```

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();
```

3.2.2 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();
```

3.2.3 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.

3.2.4 Modularising the Schema IDL

Having 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 has been split into multiple files and merged all together just before schema generation.

```
SchemaParser schemaParser = new SchemaParser();
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.parse(schemaFile1));
typeRegistry.merge(schemaParser.parse(schemaFile2));
typeRegistry.merge(schemaParser.parse(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
}
```

3.2.5 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
}
```

3.3 Execution

3.3.1 Queries

To execute a query against a schema build a new `GraphQL` object with the appropriate arguments and then call `execute()`.

The result of a query is an `ExecutionResult` which is the query data and/or a list of errors.

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

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

ExecutionInput executionInput = ExecutionInput.newExecutionInput().query("query {␣
↪hero { name } }")
        .build();

ExecutionResult executionResult = graphQL.execute(executionInput);

Object data = executionResult.getData();
List<GraphQLError> errors = executionResult.getErrors();
```

More complex query examples can be found in the StarWars query tests

3.3.2 Data Fetchers

Each graphql field type has a `graphql.schema.DataFetcher` associated with it. Other graphql implementations often call this type of code *resolvers**.

Often you can rely on `graphql.schema.PropertyDataFetcher` to examine Java POJO objects to provide field values from them. If your don't specify a data fetcher on a field, this is what will be used.

However you will need to fetch your top level domain objects via your own custom data fetchers. This might involve making a database call or contacting another system over HTTP say.

`graphql-java` is not opinionated about how you get your domain data objects, that is very much your concern. It is also not opinionated on user authorisation to that data. You should push all that logic into your business logic layer code.

A data fetcher might look like this:

```
DataFetcher userDataFetcher = new DataFetcher() {
    @Override
    public Object get(DataFetchingEnvironment environment) {
        return fetchUserFromDatabase(environment.getArgument("userId"));
    }
};
```

Each `DataFetcher` is passed a `graphql.schema.DataFetchingEnvironment` object which contains what field is being fetched, what arguments have been supplied to the field and other information such as the field's parent object, the query root object or the query context object.

In the above example, the execution will wait for the data fetcher to return before moving on. You can make execution of the `DataFetcher` asynchronous by returning a `CompletionStage` to data, that is explained more further

down this page.

3.3.3 Exceptions while fetching data

If an exception happens during the data fetcher call, then the execution strategy by default will make a `graphql.ExceptionWhileDataFetching` error and add it to the list of errors on the result. Remember graphql allows partial results with errors.

Here is the code for the standard behaviour.

```
public class SimpleDataFetcherExceptionHandler implements DataFetcherExceptionHandler
↪{
    private static final Logger log = LoggerFactory.
↪getLogger(SimpleDataFetcherExceptionHandler.class);

    @Override
    public void accept(DataFetcherExceptionHandlerParameters handlerParameters) {
        Throwable exception = handlerParameters.getException();
        SourceLocation sourceLocation = handlerParameters.getField().
↪getSourceLocation();
        ExecutionPath path = handlerParameters.getPath();

        ExceptionWhileDataFetching error = new ExceptionWhileDataFetching(path,␣
↪exception, sourceLocation);
        handlerParameters.getExecutionContext().addError(error);
        log.warn(error.getMessage(), exception);
    }
}
```

If the exception you throw is itself a *GraphqlError* then it will transfer the message and custom extensions attributes from that exception into the *ExceptionWhileDataFetching* object. This allows you to place your own custom attributes into the graphql error that is sent back to the caller.

For example imagine your data fetcher threw this exception. The *foo* and *fizz* attributes would be included in the resultant graphql error.

```
class CustomRuntimeException extends RuntimeException implements GraphQLError {
    @Override
    public Map<String, Object> getExtensions() {
        Map<String, Object> customAttributes = new LinkedHashMap<>();
        customAttributes.put("foo", "bar");
        customAttributes.put("fizz", "whizz");
        return customAttributes;
    }

    @Override
    public List<SourceLocation> getLocations() {
        return null;
    }

    @Override
    public ErrorType getErrorType() {
        return ErrorType.DataFetchingException;
    }
}
```

You can change this behaviour by creating your own `graphql.execution.DataFetcherExceptionHandler` exception handling code and giving that to the execution strategy.

For example the code above records the underlying exception and stack trace. Some people may prefer not to see that in the output error list. So you can use this mechanism to change that behaviour.

```
DataFetcherExceptionHandler handler = new DataFetcherExceptionHandler() {
    @Override
    public void accept(DataFetcherExceptionHandlerParameters handlerParameters) {
        //
        // do your custom handling here.  The parameters have all you need
    }
};
ExecutionStrategy executionStrategy = new AsyncExecutionStrategy(handler);
```

3.3.4 Serializing results to JSON

The most common way to call graphql is over HTTP and to expect a JSON response back. So you need to turn an *graphql.ExecutionResult* into a JSON payload.

A common way to do that is use a JSON serialisation library like Jackson or GSON. However exactly how they interpret the data result is particular to them. For example *nulls* are important in graphql results and hence you must set up the json mappers to include them.

To ensure you get a JSON result that confirms 100% to the graphql spec, you should call *toSpecification* on the result and then send that back as JSON.

This will ensure that the result follows the specification outlined in http://facebook.github.io/graphql/#sec-Response

```
ExecutionResult executionResult = graphQL.execute(executionInput);

Map<String, Object> toSpecificationResult = executionResult.toSpecification();

sendAsJson(toSpecificationResult);
```

3.3.5 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 :

```
GraphQLInputObjectType episodeType = GraphQLInputObjectType.newInputObject()
        .name("Episode")
        .field(newInputObjectField()
                .name("episodeNumber")
                .type(Scalars.GraphQLInt))
```

```
        .build();

GraphQLInputObjectType reviewInputType = GraphQLInputObjectType.newInputObject()
        .name("ReviewInput")
        .field(newInputObjectField()
                .name("stars")
                .type(Scalars.GraphQLString))
        .field(newInputObjectField()
                .name("commentary")
                .type(Scalars.GraphQLString))
        .build();

GraphQLObjectType reviewType = newObject()
        .name("Review")
        .field(newFieldDefinition()
                .name("stars")
                .type(GraphQLString))
        .field(newFieldDefinition()
                .name("commentary")
                .type(GraphQLString))
        .build();

GraphQLObjectType createReviewForEpisodeMutation = newObject()
        .name("CreateReviewForEpisodeMutation")
        .field(newFieldDefinition()
                .name("createReview")
                .type(reviewType)
                .argument(newArgument()
                        .name("episode")
                        .type(episodeType)
                )
                .argument(newArgument()
                        .name("review")
                        .type(reviewInputType)
                )
                .dataFetcher(mutationDataFetcher())
        )
        .build();

GraphQLSchema schema = GraphQLSchema.newSchema()
        .query(queryType)
        .mutation(createReviewForEpisodeMutation)
        .build();
```

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) {
            //
            // The graphql specification dictates that input object arguments MUST
            // be maps.  You can convert them to POJOs inside the data fetcher if that
            // suits your code better
```

```
            //
            // See http://facebook.github.io/graphql/October2016/#sec-Input-Objects
            //
            Map<String, Object> episodeInputMap = environment.getArgument("episode");
            Map<String, Object> reviewInputMap = environment.getArgument("review");

            //
            // in this case we have type safe Java objects to call our backing code␣
↪with
            //
            EpisodeInput episodeInput = EpisodeInput.fromMap(episodeInputMap);
            ReviewInput reviewInput = ReviewInput.fromMap(reviewInputMap);

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

            // 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.

3.3.6 Asynchronous Execution

graphql-java uses fully asynchronous execution techniques when it executes queries. You can get the `CompleteableFuture` to results by calling `executeAsync()` like this

```
GraphQL graphQL = buildSchema();

ExecutionInput executionInput = ExecutionInput.newExecutionInput().query("query {␣
↪hero { name } }")
        .build();

CompletableFuture<ExecutionResult> promise = graphQL.executeAsync(executionInput);

promise.thenAccept(executionResult -> {
    // here you might send back the results as JSON over HTTP
    encodeResultToJsonAndSendResponse(executionResult);
});

promise.join();
```

The use of `CompletableFuture` allows you to compose actions and functions that will be applied when the execution completes. The final call to `.join()` waits for the execution to happen.

In fact under the covers, the graphql-java engine uses asynchronous execution and makes the `.execute()` method appear synchronous by calling join for you. So the following code is in fact the same.

```
ExecutionResult executionResult = graphQL.execute(executionInput);

// the above is equivalent to the following code (in long hand)

CompletableFuture<ExecutionResult> promise = graphQL.executeAsync(executionInput);
ExecutionResult executionResult2 = promise.join();
```

If a `graphql.schema.DataFetcher` returns a `CompletableFuture<T>` object then this will be composed into the overall asynchronous query execution. This means you can fire off a number of field fetching requests in parallel. Exactly what threading strategy you use is up to your data fetcher code.

The following code uses the standard Java `java.util.concurrent.ForkJoinPool.commonPool()` thread executor to supply values in another thread.

```
DataFetcher userDataFetcher = new DataFetcher() {
    @Override
    public Object get(DataFetchingEnvironment environment) {
        CompletableFuture<User> userPromise = CompletableFuture.supplyAsync(() -> {
            return fetchUserViaHttp(environment.getArgument("userId"));
        });
        return userPromise;
    }
};
```

The code above is written in long form. With Java 8 lambdas it can be written more succinctly as follows

```
DataFetcher userDataFetcher = environment -> CompletableFuture.supplyAsync(
        () -> fetchUserViaHttp(environment.getArgument("userId")));
```

The graphql-java engine ensures that all the `CompletableFuture` objects are composed together to provide an execution result that follows the graphql specification.

There is a helpful shortcut in graphql-java to create asynchronous data fetchers. Use `graphql.schema.AsyncDataFetcher.async(DataFetcher<T>)` to wrap a `DataFetcher`. This can be used with static imports to produce more readable code.

```
DataFetcher userDataFetcher = async(environment -> fetchUserViaHttp(environment.
↪getArgument("userId")));
```

3.3.7 Execution Strategies

A class derived from `graphql.execution.ExecutionStrategy` is used to run a query or mutation. A number of different strategies are provided with graphql-java and if you are really keen you can even write your own.

You can wire in what execution strategy to use when you create the `GraphQL` object.

```
GraphQL.newGraphQL(schema)
        .queryExecutionStrategy(new AsyncExecutionStrategy())
        .mutationExecutionStrategy(new AsyncSerialExecutionStrategy())
        .build();
```

In fact the code above is equivalent to the default settings and is a very sensible choice of execution strategies for most cases.

AsyncExecutionStrategy

By default the "query" execution strategy is `graphql.execution.AsyncExecutionStrategy` which will dispatch each field as `CompleteableFuture` objects and not care which ones complete first. This strategy allows for the most performant execution.

The data fetchers invoked can themselves return *CompletionStage'* values and this will create fully asynchronous behaviour.

So imagine a query as follows

```
query {
  hero {
    enemies {
      name
    }
    friends {
      name
    }
  }
}
```

The `AsyncExecutionStrategy` is free to dispatch the *enemies* field at the same time as the *friends* field. It does not have to do *enemies* first followed by *friends*, which would be less efficient.

It will however assemble the results in order. The query result will follow the graphql specification and return object values assembled in query field order. Only the execution of data fetching is free to be in any order.

This behaviour is allowed in the graphql specification and in fact is actively encouraged http://facebook.github.io/graphql/#sec-Query for read only queries.

See specification for details.

AsyncSerialExecutionStrategy

The graphql specification says that mutations MUST be executed serially and in the order in which the query fields occur.

So `graphql.execution.AsyncSerialExecutionStrategy` is used by default for mutations and will ensure that each field is completed before it processes the next one and so forth. You can still return `CompletionStage` objects in the mutation data fetchers, however they will be executed serially and will be completed before the next mutation field data fetcher is dispatched.

ExecutorServiceExecutionStrategy

The `graphql.execution.ExecutorServiceExecutionStrategy` execution strategy will always dispatch each field fetch in an asynchronous manner, using the executor you give it. It differs from `AsyncExecutionStrategy` in that it does not rely on the data fetchers to be asynchronous but rather makes the field fetch invocation asynchronous by submitting each field to the provided *java.util.concurrent.ExecutorService*.

This behaviour makes it unsuitable to be used as a mutation execution strategy.

```
ExecutorService  executorService = 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.newGraphQL(StarWarsSchema.starWarsSchema)
        .queryExecutionStrategy(new ExecutorServiceExecutionStrategy(executorService))
        .mutationExecutionStrategy(new AsyncSerialExecutionStrategy())
        .build();
```

SubscriptionExecutionStrategy

Graphql subscriptions allows you to create stateful subscriptions to graphql data. You uses `SubscriptionExecutionStrategy` as your execution strategy as it has the support for the reactive-streams APIs.

See http://www.reactive-streams.org/ for more information on the reactive `Publisher` and `Subscriber` interfaces.

Also see the page on subscriptions for more details on how to write a subscription based graphql service.

BatchedExecutionStrategy

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

on how BatchedExecutionStrategy works here. Its a pretty special case that I don't know how to explain properly

3.3.8 Limiting Field Visibility

By default every fields defined in a *GraphqlSchema* is available. There are cases where you may want to restrict certain fields depending on the user.

You can do this by using a *graphql.schema.visibility.GraphqlFieldVisibility* implementation and attaching it to the schema.

A simple *graphql.schema.visibility.BlockedFields* implementation based on fully qualified field name is provided.

```
GraphqlFieldVisibility blockedFields = BlockedFields.newBlock()
        .addPattern("Character.id")
        .addPattern("Droid.appearsIn")
        .addPattern(".*\\.hero") // it uses regular expressions
        .build();

GraphQLSchema schema = GraphQLSchema.newSchema()
        .query(StarWarsSchema.queryType)
        .fieldVisibility(blockedFields)
        .build();
```

There is also another implementation that prevents instrumentation from being able to be performed on your schema, if that is a requirement.

Note that this puts your server in contravention of the graphql specification and expectations of most clients so use this with caution.

```
GraphQLSchema schema = GraphQLSchema.newSchema()
        .query(StarWarsSchema.queryType)
        .fieldVisibility(NoIntrospectionGraphqlFieldVisibility.NO_INTROSPECTION_FIELD_
↪VISIBILITY)
        .build();
```

You can create your own derivation of *GraphqlFieldVisibility* to check what ever you need to do to work out what fields should be visible or not.

```
class CustomFieldVisibility implements GraphqlFieldVisibility {

    final YourUserAccessService userAccessService;
```

```
    CustomFieldVisibility(YourUserAccessService userAccessService) {
        this.userAccessService = userAccessService;
    }

    @Override
    public List<GraphQLFieldDefinition> getFieldDefinitions(GraphQLFieldsContainer␣
↪fieldsContainer) {
        if ("AdminType".equals(fieldsContainer.getName())) {
            if (!userAccessService.isAdminUser()) {
                return Collections.emptyList();
            }
        }
        return fieldsContainer.getFieldDefinitions();
    }

    @Override
    public GraphQLFieldDefinition getFieldDefinition(GraphQLFieldsContainer␣
↪fieldsContainer, String fieldName) {
        if ("AdminType".equals(fieldsContainer.getName())) {
            if (!userAccessService.isAdminUser()) {
                return null;
            }
        }
        return fieldsContainer.getFieldDefinition(fieldName);
    }
}
```

3.3.9 Query Caching

Before the `graphql-java` engine executes a query it must be parsed and validated, and this process can be somewhat time consuming.

To avoid the need for re-parse/validate the `GraphQL.Builder` allows an instance of `PreparsedDocumentProvider` to reuse `Document` instances.

Please note that this does not cache the result of the query, only the parsed `Document`.

```
Cache<String, PreparsedDocumentEntry> cache = Caffeine.newBuilder().maximumSize(10_
↪000).build(); (1)
GraphQL graphQL = GraphQL.newGraphQL(StarWarsSchema.starWarsSchema)
        .preparsedDocumentProvider(cache::get) (2)
        .build();
```

1. Create an instance of preferred cache instance, here is Caffeine used as it is a high quality caching solution. The cache instance should be thread safe and shared.
2. The `PreparsedDocumentProvider` is a functional interface with only a get method and we can therefore pass a method reference that matches the signature into the builder.

In order to achieve high cache hit ration it is recommended that field arguments are passed in as variables instead of directly in the query.

The following query:

```
query HelloTo {
     sayHello(to: "Me") {
        greeting
```

```
    }
}
```

Should be rewritten as:

```
query HelloTo($to: String!) {
    sayHello(to: $to) {
       greeting
    }
}
```

with variables:

```
{
   "to": "Me"
}
```

The query is now reused regardless of variable values provided.

3.4 Subscriptions

3.4.1 Subscription Queries

Graphql subscriptions allow you subscribe to a reactive source and as new data arrives then a graphql query is applied over that data and the results are passed on.

See http://graphql.org/blog/subscriptions-in-graphql-and-relay/ for more general details on graphql subscriptions.

Imagine you have an stock market pricing service and you make a graphql subscription to it like this

```
subscription StockCodeSubscription {
    stockQuotes(stockCode:"IBM') {
        dateTime
        stockCode
        stockPrice
        stockPriceChange
    }
}
```

graphql subscriptions allow a stream of `ExecutionResult` objects to be sent down each time the stock price changes. The field selection set will applied to the underlying data and are represented just like any other graphql query.

What is special is that the initial result of a subscription query is a reactive-streams `Publisher` object which you need to use to get the future values.

You need to use `SubscriptionExecutionStrategy` as your execution strategy as it has the support for the reactive-streams APIs.

```
GraphQL graphQL = GraphQL
        .newGraphQL(schema)
        .subscriptionExecutionStrategy(new SubscriptionExecutionStrategy())
        .build();

ExecutionResult executionResult = graphQL.execute(query);
```

```
Publisher<ExecutionResult> stockPriceStream = executionResult.getData();
```

The `Publisher<ExecutionResult>` here is the publisher of a stream of events. You need to subscribe to this with your processing code which will look something like the following

```
GraphQL graphQL = GraphQL
        .newGraphQL(schema)
        .subscriptionExecutionStrategy(new SubscriptionExecutionStrategy())
        .build();

String query = "" +
        "    subscription StockCodeSubscription {\n" +
        "        stockQuotes(stockCode:\"IBM') {\n" +
        "            dateTime\n" +
        "            stockCode\n" +
        "            stockPrice\n" +
        "            stockPriceChange\n" +
        "        }\n" +
        "    }\n";

ExecutionResult executionResult = graphQL.execute(query);

Publisher<ExecutionResult> stockPriceStream = executionResult.getData();

AtomicReference<Subscription> subscriptionRef = new AtomicReference<>();
stockPriceStream.subscribe(new Subscriber<ExecutionResult>() {

    @Override
    public void onSubscribe(Subscription s) {
        subscriptionRef.set(s);
        s.request(1);
    }

    @Override
    public void onNext(ExecutionResult er) {
        //
        // process the next stock price
        //
        processStockPriceChange(er.getData());

        //
        // ask the publisher for one more item please
        //
        subscriptionRef.get().request(1);
    }

    @Override
    public void onError(Throwable t) {
        //
        // The upstream publishing data source has encountered an error
        // and the subscription is now terminated.  Real production code needs
        // to decide on a error handling strategy.
        //
    }

    @Override
    public void onComplete() {
```

```
        //
        // the subscription has completed.  There is not more data
        //
    }
});
```

You are now writing reactive-streams code to consume a series of `ExecutionResults`. You can see more details on reactive-streams code here http://www.reactive-streams.org/

`RxJava` is a popular implementation of reactive-streams. Check out http://reactivex.io/intro.html to find out more about creating Publishers of data and Subscriptions to that data.

graphql-java only produces a stream of results. It does not concern itself with sending these over the network on things like web sockets and so on. That is important but not a concern of the base graphql-java library.

We have put together a basic example of using websockets (backed by Jetty) with a simulated stock price application that is built using RxJava.

See https://github.com/graphql-java/graphql-java-subscription-example for more detailed code on handling network concerns and the like.

3.4.2 Subscription Data Fetchers

The `DataFetcher` behind a subscription field is responsible for creating the `Publisher` of data. The objects return by this Publisher will be mapped over the graphql query as each arrives and then sent back out as an execution result.

You data fetcher is going to look something like this.

```
DataFetcher<Publisher<StockInfo>> publisherDataFetcher = new DataFetcher<Publisher
↪<StockInfo>>() {
    @Override
    public Publisher<StockInfo> get(DataFetchingEnvironment environment) {
        String stockCodeArg = environment.getArgument("stockCode");
        return buildPublisherForStockCode(stockCodeArg);
    }
};
```

Now the exact details of how you get that stream of events is up to you and you're reactive code. graphql-java gives you a way to map the graphql query fields over that stream of objects just like a standard graphql query.

3.5 Runtime Exceptions

Runtime exceptions can be thrown by the graphql engine if certain exceptional situations are encountered. The following are a list of the exceptions that can be thrown all the way out of a `graphql.execute(...)` call.

These are not graphql errors in execution but rather totally unacceptable conditions in which to execute a graphql query.

- *graphql.schema.CoercingSerializeException*

  is thrown when a value cannot be serialised by a Scalar type, for example a String value being coerced as an Int.

- *graphql.schema.CoercingParseValueException*

is thrown when a value cannot be parsed by a Scalar type, for example a String input value being parsed as an Int.

- *graphql.execution.UnresolvedTypeException*

is thrown if a graphql.schema.TypeResolver‘ fails to provide a concrete object type given a interface or union type.

- *graphql.execution.NonNullableValueCoercedAsNullException*

is thrown if a non null variable argument is coerced as a null value during execution.

- *graphql.execution.InputMapDefinesTooManyFieldsException*

is thrown if a map used for an input type object contains more keys than is defined in that input type.

- *graphql.schema.validation.InvalidSchemaException*

is thrown if the schema is not valid when built via graphql.schema.GraphQLSchema.Builder#build()‘

- *graphql.GraphQLException*

is thrown as a general purpose runtime exception, for example if the code cant access a named field when examining a POJO, it is analogous to a RuntimeException if you will.

- *graphql.AssertException*

is thrown as a low level code assertion exception for truly unexpected code conditions, things we assert should never happen in practice.

3.6 Using Dataloader

If you are using `graphql`, you are likely to making queries on a graph of data (surprise surprise). But its easy to implement inefficient code with naive loading of a graph of data.

Using `java-dataloader` will help you to make this a more efficient process by both caching and batching requests for that graph of data items. If `dataloader` has seen a data item before, it will have cached the value and will return it without having to ask for it again.

Imagine we have the StarWars query outlined below. It asks us to find a hero and their friend's names and their friend's friend's names. It is likely that many of these people will be friends in common.

```
{
    hero {
        name
        friends {
            name
            friends {
               name
            }
        }
    }
}
```

The result of this query is displayed below. You can see that Han, Leia, Luke and R2-D2 are a tight knit bunch of friends and share many friends in common.

```
[hero: [name: 'R2-D2', friends: [
        [name: 'Luke Skywalker', friends: [
                [name: 'Han Solo'], [name: 'Leia Organa'], [name: 'C-3PO'], [name:
↪'R2-D2']]],
```

```
        [name: 'Han Solo', friends: [
                [name: 'Luke Skywalker'], [name: 'Leia Organa'], [name: 'R2-D2']]],
        [name: 'Leia Organa', friends: [
                [name: 'Luke Skywalker'], [name: 'Han Solo'], [name: 'C-3PO'], [name:
↪'R2-D2']]]]
]
```

A naive implementation would called a *DataFetcher* to retrieved a person object every time it was invoked.

In this case it would be *15* calls over the network. Even though the group of people have a lot of common friends. With *dataloader* you can make the *graphql* query much more efficient.

As *graphql* descends each level of the query (eg as it processes *hero* and then *friends* and then for each their *friends*), the data loader is called to "promise" to deliver a person object. At each level *dataloader.dispatch()* will be called to fire off the batch requests for that part of the query. With caching turned on (the default) then any previously returned person will be returned as is for no cost.

In the above example there are only *5* unique people mentioned but with caching and batching retrieval in place their will be only *3* calls to the batch loader function. *3* calls over the network or to a database is much better than *15* calls you will agree.

If you use capabilities like *java.util.concurrent.CompletableFuture.supplyAsync()* then you can make it even more efficient by making the the remote calls asynchronous to the rest of the query. This will make it even more timely since multiple calls can happen at once if need be.

Here is how you might put this in place:

```
// a batch loader function that will be called with N or more keys for batch loading
BatchLoader<String, Object> characterBatchLoader = new BatchLoader<String, Object>() {
    @Override
    public CompletionStage<List<Object>> load(List<String> keys) {
        //
        // we use supplyAsync() of values here for maximum parellisation
        //
        return CompletableFuture.supplyAsync(() ->␣
↪getCharacterDataViaBatchHTTPApi(keys));
    }
};

// a data loader for characters that points to the character batch loader
DataLoader<String, Object> characterDataLoader = new DataLoader<>
↪(characterBatchLoader);

//
// use this data loader in the data fetchers associated with characters and put them␣
↪into
// the graphql schema (not shown)
//
DataFetcher heroDataFetcher = new DataFetcher() {
    @Override
    public Object get(DataFetchingEnvironment environment) {
        return characterDataLoader.load("2001"); // R2D2
    }
};

DataFetcher friendsDataFetcher = new DataFetcher() {
    @Override
    public Object get(DataFetchingEnvironment environment) {
        StarWarsCharacter starWarsCharacter = environment.getSource();
```

```
        List<String> friendIds = starWarsCharacter.getFriendIds();
        return characterDataLoader.loadMany(friendIds);
    }
};

//
// DataLoaderRegistry is a place to register all data loaders in that needs to be␣
↪dispatched together
// in this case there is 1 but you can have many
//
DataLoaderRegistry registry = new DataLoaderRegistry();
registry.register("character", characterDataLoader);

//
// this instrumentation implementation will dispatch all the dataloaders
// as each level fo the graphql query is executed and hence make batched objects
// available to the query and the associated DataFetchers
//
DataLoaderDispatcherInstrumentation dispatcherInstrumentation
        = new DataLoaderDispatcherInstrumentation(registry);

//
// now build your graphql object and execute queries on it.
// the data loader will be invoked via the data fetchers on the
// schema fields
//
GraphQL graphQL = GraphQL.newGraphQL(buildSchema())
        .instrumentation(dispatcherInstrumentation)
        .build();
```

```
```

One thing to note is the above only works if you use *DataLoaderDispatcherInstrumentation* which makes sure *dataLoader.dispatch()* is called. If this was not in place, then all the promises to data will never be dispatched ot the batch loader function and hence nothing would ever resolve.

3.6.1 Per Request Data Loaders

If you are serving web requests then the data can be specific to the user requesting it. If you have user specific data then you will not want to cache data meant for user A to then later give it to user B in a subsequent request.

The scope of your DataLoader instances is important. You might want to create them per web request to ensure data is only cached within that web request and no more.

If your data can be shared across web requests then you might want to scope your data loaders so they survive longer than the web request say.

But if you are doing per request data loaders then creating a new set of `GraphQL` and `DataLoader` objects per request is super cheap. Its the `GraphQLSchema` creation that can be expensive, especially if you are using graphql IDL parsing.

Structure your code so that the schema is statically held, perhaps in a static variable or in a singleton IoC component but build out a new `GraphQL` set of objects on each request.

```
GraphQLSchema staticSchema = staticSchema_Or_MayBeFrom_IoC_Injection();

DataLoaderRegistry registry = new DataLoaderRegistry();
registry.register("character", getCharacterDataLoader());
```

```
DataLoaderDispatcherInstrumentation dispatcherInstrumentation
        = new DataLoaderDispatcherInstrumentation(registry);

GraphQL graphQL = GraphQL.newGraphQL(staticSchema)
        .instrumentation(dispatcherInstrumentation)
        .build();

graphQL.execute("{ helloworld }");

// you can now throw away the GraphQL and hence DataLoaderDispatcherInstrumentation
// and DataLoaderRegistry objects since they are really cheap to build per request
```

3.7 Instrumentation

The `graphql.execution.instrumentation.Instrumentation` interface allows you to inject code that can observe the execution of a query and also change the runtime behaviour.

The primary use case for this is to allow say performance monitoring and custom logging but it could be used for many different purposes.

When you build the `Graphql` object you can specify what `Instrumentation` to use (if any).

```
GraphQL.newGraphQL(schema)
        .instrumentation(new TracingInstrumentation())
        .build();
```

3.7.1 Custom Instrumentation

An implementation of `Instrumentation` needs to implement the "begin" step methods that represent the execution of a graphql query.

Each step must give back a non null `graphql.execution.instrumentation.InstrumentationContext` object which will be called back when the step completes, and will be told that it succeeded or failed with a Throwable.

The following is a basic custom `Instrumentation` that measures overall execution time and puts it into a stateful object.

```
class CustomInstrumentationState implements InstrumentationState {
    private Map<String, Object> anyStateYouLike = new HashMap<>();

    void recordTiming(String key, long time) {
        anyStateYouLike.put(key, time);
    }
}

class CustomInstrumentation implements Instrumentation {
    @Override
    public InstrumentationState createState() {
        //
        // instrumentation state is passed during each invocation of an␣
↪Instrumentation method
        // and allows you to put stateful data away and reference it during the query␣
↪execution
```

```
        //
        return new CustomInstrumentationState();
    }

    @Override
    public InstrumentationContext<ExecutionResult>␣
↪beginExecution(InstrumentationExecutionParameters parameters) {
        long startNanos = System.nanoTime();
        return (result, throwable) -> {

            CustomInstrumentationState state = parameters.getInstrumentationState();
            state.recordTiming(parameters.getQuery(), System.nanoTime() - startNanos);
        };
    }

    @Override
    public InstrumentationContext<Document>␣
↪beginParse(InstrumentationExecutionParameters parameters) {
        //
        // You MUST return a non null object but it does not have to do anything and␣
↪hence
        // you use this class to return a no-op object
        //
        return new NoOpInstrumentation.NoOpInstrumentationContext<>();
    }

    @Override
    public InstrumentationContext<List<ValidationError>>␣
↪beginValidation(InstrumentationValidationParameters parameters) {
        return new NoOpInstrumentation.NoOpInstrumentationContext<>();
    }

    @Override
    public InstrumentationContext<ExecutionResult>␣
↪beginDataFetch(InstrumentationDataFetchParameters parameters) {
        return new NoOpInstrumentation.NoOpInstrumentationContext<>();
    }

    @Override
    public InstrumentationContext<CompletableFuture<ExecutionResult>>␣
↪beginExecutionStrategy(InstrumentationExecutionStrategyParameters parameters) {
        return new NoOpInstrumentation.NoOpInstrumentationContext<>();
    }

    @Override
    public InstrumentationContext<ExecutionResult>␣
↪beginField(InstrumentationFieldParameters parameters) {
        return new NoOpInstrumentation.NoOpInstrumentationContext<>();
    }

    @Override
    public InstrumentationContext<Object>␣
↪beginFieldFetch(InstrumentationFieldFetchParameters parameters) {
        return new NoOpInstrumentation.NoOpInstrumentationContext<>();
    }

    @Override
    public DataFetcher<?> instrumentDataFetcher(DataFetcher<?> dataFetcher,␣
↪InstrumentationFieldFetchParameters parameters) {
```

```
        //
        // this allows you to intercept the data fetcher used ot fetch a field and␣
↪provide another one, perhaps
        // that enforces certain behaviours or has certain side effects on the data
        //
        return dataFetcher;
    }

    @Override
    public CompletableFuture<ExecutionResult>␣
↪instrumentExecutionResult(ExecutionResult executionResult,␣
↪InstrumentationExecutionParameters parameters) {
        //
        // this allows you to instrument the execution result some how.  For example␣
↪the Tracing support uses this to put
        // the `extensions` map of data in place
        //
        return CompletableFuture.completedFuture(executionResult);
    }
}
```

3.7.2 Chaining Instrumentation

You can combine multiple `Instrumentation` objects together using the `graphql.execution.instrumentation.ChainedInstrumentation` class which accepts a list of `Instrumentation` objects and calls them in that defined order.

```
List<Instrumentation> chainedList = new ArrayList<>();
chainedList.add(new FooInstrumentation());
chainedList.add(new BarInstrumentation());
ChainedInstrumentation chainedInstrumentation = new␣
↪ChainedInstrumentation(chainedList);

GraphQL.newGraphQL(schema)
        .instrumentation(chainedInstrumentation)
        .build();
```

3.7.3 Apollo Tracing Instrumentation

`graphql.execution.instrumentation.tracing.TracingInstrumentation` is an `Instrumentation` implementation that creates tracing information about the query that is being executed.

It follows the Apollo proposed tracing format defined at https://github.com/apollographql/apollo-tracing

A detailed tracing map will be created and placed in the `extensions` section of the result.

So given a query like

```
query {
  hero {
    name
    friends {
      name
    }
  }
}
```

It would return a result like


```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  },
  "extensions": {
    "tracing": {
      "version": 1,
      "startTime": "2017-08-14T23:13:39.362Z",
      "endTime": "2017-08-14T23:13:39.497Z",
      "duration": 135589186,
      "execution": {
        "resolvers": [
          {
            "path": [
              "hero"
            ],
            "parentType": "Query",
            "returnType": "Character",
            "fieldName": "hero",
            "startOffset": 105697585,
            "duration": 79111240
          },
          {
            "path": [
              "hero",
              "name"
            ],
            "parentType": "Droid",
            "returnType": "String",
            "fieldName": "name",
            "startOffset": 125010028,
            "duration": 20213
          },
          {
            "path": [
              "hero",
              "friends"
            ],
            "parentType": "Droid",
            "returnType": "[Character]",
            "fieldName": "friends",
            "startOffset": 133352819,
```

```
            "duration": 7927560
          },
          {
            "path": [
              "hero",
              "friends",
              0,
              "name"
            ],
            "parentType": "Human",
            "returnType": "String",
            "fieldName": "name",
            "startOffset": 134105887,
            "duration": 6783
          },
          {
            "path": [
              "hero",
              "friends",
              1,
              "name"
            ],
            "parentType": "Human",
            "returnType": "String",
            "fieldName": "name",
            "startOffset": 134725922,
            "duration": 7016
          },
          {
            "path": [
              "hero",
              "friends",
              2,
              "name"
            ],
            "parentType": "Human",
            "returnType": "String",
            "fieldName": "name",
            "startOffset": 134875089,
            "duration": 6342
          }
        ]
      }
    }
  }
}
```


3.7.4 Field Validation Instrumentation

`graphql.execution.instrumentation.fieldvalidation.FieldValidationInstrumentation` is an `Instrumentation` implementation that can be used to validate fields and their arguments before query execution. If errors are returned during this process then the query execution is aborted and the errors will be in the query result.

You can make you own custom implementation of `FieldValidation` or you can use the `SimpleFieldValidation` class to add simple per field checks rules.

```
ExecutionPath fieldPath = ExecutionPath.parse("/user");
FieldValidation fieldValidation = new SimpleFieldValidation()
        .addRule(fieldPath, new BiFunction<FieldAndArguments,␣
↪FieldValidationEnvironment, Optional<GraphQLError>>() {
            @Override
            public Optional<GraphQLError> apply(FieldAndArguments fieldAndArguments,␣
↪FieldValidationEnvironment environment) {
                String nameArg = fieldAndArguments.getFieldArgument("name");
                if (nameArg.length() > 255) {
                    return Optional.of(environment.mkError("Invalid user name",␣
↪fieldAndArguments));
                }
                return Optional.empty();
            }
        });

FieldValidationInstrumentation instrumentation = new FieldValidationInstrumentation(
        fieldValidation
);

GraphQL.newGraphQL(schema)
        .instrumentation(instrumentation)
        .build();
```

3.8 Relay Support

Very basic support for Relay is included.

Note: Relay refers here to "Relay Classic", there is no support for "Relay Modern".

Please look at https://github.com/graphql-java/todomvc-relay-java for a full example project,

Relay send queries to the GraphQL server as JSON containing a `query` field and a `variables` field. The `query` field is a JSON string, and the `variables` field is a map of variable definitions. A relay-compatible server will need to parse this JSON and pass the `query` string to this library as the query and the `variables` map as the third argument to `execute` as shown below.

```
@RequestMapping(value = "/graphql", method = RequestMethod.POST, produces = MediaType.
↪APPLICATION_JSON_VALUE)
@ResponseBody
public Object executeOperation(@RequestBody Map body) {
    String query = (String) body.get("query");
    Map<String, Object> variables = (Map<String, Object>) body.get("variables");
    if (variables == null) {
        variables = new LinkedHashMap<>();
    }
    ExecutionResult executionResult = graphql.execute(query, (Object) null,␣
↪variables);
    Map<String, Object> result = new LinkedHashMap<>();
    if (executionResult.getErrors().size() > 0) {
        result.put("errors", executionResult.getErrors());
        log.error("Errors: {}", executionResult.getErrors());
    }
    result.put("data", executionResult.getData());
    return result;
}
```

3.8.1 Apollo Support

There is no special support for Apollo included: Apollo works with any schema.

The Controller example shown above works with Apollo too.

3.9 Logging

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

3.10 Application concerns

The graphql-java library concentrates on providing an engine for the execution of queries according to the specification.

It does not concern itself about other high level application concerns such as the following :

- Database access
- Caching data
- Data authorisation
- Data pagination
- HTTP transfer
- JSON endoding
- Code wiring via dependency injection

You need to push these concerns into your business logic layers.

The following are great links to read more about this

- http://graphql.org/learn/serving-over-http/
- http://graphql.org/learn/authorization/
- http://graphql.org/learn/pagination/
- http://graphql.org/learn/caching/

3.10.1 Context Objects

You can pass in a context object during query execution that will allow you to better invoke that business logic.

For example the edge of your application could be performing user detection and you need that information inside the graphql execution to perform authorisation.

This made up example shows how you can pass yourself information to help execute your queries.

```
//
// this could be code that authorises the user in some way and sets up enough context
// that can be used later inside data fetchers allowing them
// to do their job
//
UserContext contextForUser = YourGraphqlContextBuilder.
↪getContextForUser(getCurrentUser());
```

```
ExecutionInput executionInput = ExecutionInput.newExecutionInput()
        .context(contextForUser)
        .build();

ExecutionResult executionResult = graphQL.execute(executionInput);

// ...
//
// later you are able to use this context object when a data fetcher is invoked
//

DataFetcher dataFetcher = new DataFetcher() {
    @Override
    public Object get(DataFetchingEnvironment environment) {
        UserContext userCtx = environment.getContext();
        Long businessObjId = environment.getArgument("businessObjId");

        return invokeBusinessLayerMethod(userCtx, businessObjId);
    }
};
```

3.11 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.

3.11.1 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
```