Graphene Documentation

Release 1.0

Syrus Akbary

Mar 08, 2018

Contents

Contents:

CHAPTER 1

Getting started

For an introduction to GraphQL and an overview of its concepts, please refer to the official introduction.

Let's build a basic GraphQL schema from scratch.

Requirements

- Node.js
- Graphene-JS

Project setup

```
npm install graphene-js
# or
yarn add graphene-js
```

Creating a basic Schema

A GraphQL schema describes your data model, and provides a GraphQL server with an associated set of resolve methods that know how to fetch data.

We are going to create a very simple schema, with a `Query` with only one field: `hello` and an input name. And when we query it, it should return `"Hello {name}"`.

```
import { ObjectType, Field, Schema } from "graphene-js";

@ObjectType()
class Query {
    @Field(String, {args: {name: String}})
```

```
    hello({name}) {
        return `Hello ${name || "stranger"}`;
    }
}

schema = new Schema({query: Query})
```

Querying

Then we can start querying our schema:

```
var result = await schema.execute('{ hello }')
console.log(result.data.hello) # "Hello stranger"
```

Congrats! You got your first graphene schema working!

CHAPTER 2

Types Reference

Enums

A `Enum` is a special `GraphQL` type that represents a set of symbolic names (members) bound to unique, constant values.

Definition

You can create an `Enum` using classes:

```
import { EnumType } from "graphene-js";

@EnumType()
class Episode {
    static NEWHOPE = 4
    static EMPIRE = 5
    static JEDI = 6
}
```

Graphene will automatically search for the static variables in the Enum and expose them as the enum values.

Value descriptions

It's possible to add a description to an enum value, for that the enum value needs to have the ‘‘description‘‘decorator on it.

```
@EnumType()
class Episode {
    @description("New hope episode")
    static NEWHOPE = 4

    @description("Empire episode")
```

```
    static EMPIRE = 5

    @description("JEDI episode")
    static JEDI = 6
}
```

Scalars

All Scalar types accept the following arguments. All are optional:

Base scalars

`String`

Represents textual data, represented as UTF-8 character sequences. The String type is most often used by GraphQL to represent free-form human-readable text.

`Int`

Represents non-fractional signed whole numeric values. Int can represent values between *-(2^53 - 1)* and *2^53 - 1* since represented in JSON as double-precision floating point numbers specified by IEEE 754.

`Float`

Represents signed double-precision fractional values as specified by IEEE 754.

`Boolean`

Represents *true* or *false*.

`ID`

Represents a unique identifier, often used to refetch an object or as key for a cache. The ID type appears in a JSON response as a String; however, it is not intended to be human-readable. When expected as an input type, any string (such as *"4"*) or integer (such as *4*) input value will be accepted as an ID.

Graphene also provides custom scalars for Dates, Times, and JSON:

`graphene.Date`

Represents a Date value as specified by iso8601.

`graphene.DateTime`

Represents a DateTime value as specified by iso8601.

`graphene.Time`

Represents a Time value as specified by iso8601.

Custom scalars

You can create custom scalars for your schema. The following is an example for creating a DateTime scalar:

```
  import { GraphQLScalarType } from "graphql";

  const Date = new GraphQLScalarType({
      name: 'Date',
```

```
      description: 'Date custom scalar type',
      parseValue(value) {
          return new Date(value); // value from the client
      },
      serialize(value) {
          return value.getTime(); // value sent to the client
      },
      parseLiteral(ast) {
          if (ast.kind === Kind.INT) {
              return parseInt(ast.value, 10); // ast value is always in string format
          }
          return null;
      },
});
```

Lists and Non-Null

Object types, scalars, and enums are the only kinds of types you can define in Graphene. But when you use the types in other parts of the schema, or in your query variable declarations, you can apply additional type modifiers that affect validation of those values.

List

```
import { ObjectType, List } from "graphene-js";

@ObjectType()
class Character {
    @Field(List(String)) appearsIn;
}
```

Lists work in a similar way: We can use a type modifier to mark a type as a `List`, which indicates that this field will return a list of that type. It works the same for arguments, where the validation step will expect a list for that value.

For ease of development, we can directly use js lists with one element `[]`.

Like:

```
import { ObjectType, Field } from "graphene-js";

@ObjectType()
class Character {
    @Field([String]) appearsIn;
}
```

NonNull

```
import { ObjectType, Field, NonNull } from "graphene-js";

@ObjectType()
class Character {
    @Field(NonNull(String)) name;
}
```

Here, we're using a `String` type and marking it as Non-Null by wrapping it using the `NonNull` class. This means that our server always expects to return a non-null value for this field, and if it ends up getting a null value that will actually trigger a GraphQL execution error, letting the client know that something has gone wrong.

Interfaces

An Interface contains the essential fields that will be implemented by multiple ObjectTypes.

The basics:

- Each Interface is class decorated with `InterfaceType`.
- Each attribute decorated with `@Field` represents a GraphQL Field in the Interface.

Quick example

This example model defines a `Character` interface with a name. `Human` and `Droid` are two implementations of that interface.

```
import { InterfaceType, ObjectType, Field } from "graphene-js";

@InterfaceType()
class Character {
    @Field(String) name;
}

// Human is a Character implementation
@ObjectType({
    interfaces: [Character]
})
class Human {
    @Field(String) bornIn;
}

// Droid is a Character implementation
@ObjectType({
    interfaces: [Character]
})
class Human {
    @Field(String) function;
}
```

`name` is a field on the `Character` interface that will also exist on both the `Human` and `Droid` ObjectTypes (as those implement the `Character` interface). Each ObjectType may define additional fields.

The above types have the following representation in a schema:

```
interface Character {
  name: String
}

type Droid implements Character {
  name: String
  function: String
}
```

```
type Human implements Character {
  name: String
  bornIn: String
}
```

Unions

Union types are very similar to interfaces, but they don't get to specify any common fields between the types.

The basics:

- Each Union is a JS class decorated with `UnionType`.
- Unions don't have any fields on it, just links to the possible objecttypes.

Quick example

This example model defines several ObjectTypes with their own fields. `SearchResult` is the implementation of `Union` of this object types.

```
import { ObjectType, UnionType, Field } from "graphene-js";

class Human {
    @Field(String) name;
    @Field(String) bornIn;
}

class Droid {
    @Field(String) name;
    @Field(String) primaryFunction;
}

class Starship {
    @Field(String) name;
    @Field(Number) length;
}

const SearchResult = new UnionType({
    name: 'SearchResult',
    types: [Human, Droid, Starship]
})
```

Wherever we return a SearchResult type in our schema, we might get a Human, a Droid, or a Starship. Note that members of a union type need to be concrete object types; you can't create a union type out of interfaces or other unions.

The above types have the following representation in a schema:

```
type Droid {
  name: String
  primaryFunction: String
}

type Human {
  name: String
```

```
  bornIn: String
}

type Ship {
  name: String
  length: Int
}

union SearchResult = Human | Droid | Starship
```

ObjectTypes

An ObjectType is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're querying.

The basics:

- Each ObjectType is a Python class that inherits from `graphene.ObjectType`.
- Each attribute of the ObjectType represents a `Field`.

Quick example

This example model defines a Person, with a first and a last name:

```
import { ObjectType, Field } from "graphene-js";

@ObjectType()
class Person {
    @Field(String) firstName;
    @Field(String) lastName;
    @Field(String)
    fullName() {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

firstName and **lastName** are fields of the ObjectType. Each field is specified as a class attribute, and each attribute maps to a Field.

The above `Person` ObjectType has the following schema representation:

```
type Person {
  firstName: String
  lastName: String
  fullName: String
}
```

Resolvers

A resolver is a method that resolves certain fields within a `ObjectType`. If not specififed otherwise, the resolver of a field is the `resolve_{field_name}` method on the `ObjectType`.

By default resolvers take the arguments `args`, `context` and `info`.

Quick example

This example model defines a `Query` type, which has a reverse field that reverses the given `word` argument using the `resolve_reverse` method in the class.

```
import { ObjectType, Field } from "graphene-js";

@ObjectType()
class Query {
    @Field(String, {args: {word: String}})
    reverse({word}) {
        return (word || "").split("").reverse().join("")
    }
}
```

Instances as data containers

Graphene `ObjectTypes` can act as containers too. So with the previous example you could do:

```
peter = new Person({firstName: "", lastName: ""})

peter.firstName # prints "Peter"
peter.lastName # prints "Griffin"
```

Schema

A Schema is created by supplying the root types of each type of operation, query, mutation and subscription. A schema definition is then supplied to the validator and executor.

```
import { Schema } from "graphene-js";

const schema = new Schema({
    query: MyRootQuery,
    mutation: MyRootMutation,
})
```

Types

There are some cases where the schema cannot access all of the types that we plan to have. For example, when a field returns an `Interface`, the schema doesn't know about any of the implementations.

In this case, we need to use the `types` argument when creating the Schema.

```
const schema = new Schema({
    query: MyRootQuery,
    types=[SomeExtraType, ],
})
```

Querying

To query a schema, call the `execute` method on it.

```
await schema.execute('{ hello }')
```

CHAPTER 3

Relay

Graphene Relay integration is on the works.

Useful links

- Getting started with Relay
- Relay Global Identification Specification
- Relay Cursor Connection Specification
- Relay input Object Mutation

CHAPTER 4

Incremental adoption

Graphene-JS is designed to be adopted incrementally, that means that you will be able to use Graphene types inside of your already existing schema and viceversa.

Graphene-JS types in GraphQL

Using Graphene types with your existing GraphQL types is very easy. The module have a utility function *getGraphQLType* that you can use to retrieve the native GraphQL type behind a Graphene type.

For example:

```
import { GraphQLSchema, GraphQLObjectType } from "graphql";
import { ObjectType, Field, getGraphQLType } from "graphene-js";

// Your graphene definition
@ObjectType()
class User {
  @Field(String) name
}

// Your normal GraphLQL types
var query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    viewer: {
      // Note getGraphQLType(User) will return a GraphQLObjectType
      // that can be safely used in GraphQL types
      type: getGraphQLType(User),
    }
  }
});
```

GraphQL types in Graphene

Graphene can operate with native GraphQL types seamlessly, with no extra effort for the developer. You can use GraphQL native types directly in Graphene

For example:

```
import { ObjectType, Field } from "graphene-js";

var User = GraphQLObjectType({
  name: 'User',
  fields: {
    name: {
      type: GraphQLString,
    }
  }
});

@ObjectType()
class Query {
  // User is a native GraphQL type
  @Field(User) user;
}
```

CHAPTER 5

Integrations

- Graphene-sequelize (source)