Appier

Introduction to GraphQL

Johnson Liang
Frontend Engineer, Appier
2017.08.09

Agenda

- Fundamental parts of a GraphQL server
- Defining API shape - GraphQL schema
- Resolving object fields
- Mutative APIs
- Making requests to a GraphQL server
- Solving N+1 query problem: dataloader

Appier

graphql.org

GraphQL | A query language fo

Johnson

graphql.org

Learn Code Community Blog Spec

Search docs...

GraphQL

Describe your data

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

Ask for what you want

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

Get predictable results

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

Get Started

Learn More

Appier

[Server]
API shape;
GraphQL Schema

[Client]
Query;
GraphQL

[Client]
Result

GraphQL | A query language fo

graphql.org

Johnson

Learn Code Community Blog Spec

Search docs...

GraphQL

Describe your data

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

Ask for what you want

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

Get predictable results

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

Get Started

Learn More

Appier

Fundamental parts of a GraphQL server

Appier

Runnable GraphQL server code

```javascript
const {
  graphql, GraphQLSchema, GraphQLObjectType,
  GraphQLString,
} = require('graphql');

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() {
          return (new Date).toLocaleString();
        },
      },
    },
  })
});

const query = `query { serverTime }`;

graphql(schema, query).then(result =>{
  console.log(result.data);
});

// Prints:
// {serverTime: "9/5/2016, 6:28:46 PM"}
```

```python
import graphene
from datetime import datetime


class Query(graphene.ObjectType):
  server_time = graphene.Field(graphene.String)

  def resolve_server_time(obj, args, context, info):
    return str(datetime.now())

schema = graphene.Schema(query=Query)


query = 'query { serverTime }'

result = schema.execute(query)
print(result.data)

#: Prints: OrderedDict([('serverTime', '2017-07-24
19:10:38.127089')])
```

Appier

Runnable GraphQL server code / Defining API shape (GraphQL schema)

```javascript
const {
  graphql, GraphQLSchema, GraphQLObjectType,
  GraphQLString,
} = require('graphql');
```

```python
import graphene
from datetime import datetime
```

```javascript
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() {
          return (new Date).toLocaleString();
        },
      },
    },
  })
});
```

```python
class Query(graphene.ObjectType):
  server_time = graphene.Field(graphene.String)

  def resolve_server_time(obj, args, context, info):
    return str(datetime.now())

schema = graphene.Schema(query=Query)
```

```javascript
const query = `query { serverTime }`;

graphql(schema, query).then(result =>{
  console.log(result.data);
});

// Prints:
// {serverTime: "9/5/2016, 6:28:46 PM"}
```

```python
query = 'query { serverTime }'

result = schema.execute(query)
print(result.data)

#: Prints: OrderedDict([('serverTime', '2017-07-24
19:10:38.127089')])
```

Appier

Runnable GraphQL server code / Query in GraphQL

```javascript
const {
  graphql, GraphQLSchema, GraphQLObjectType,
  GraphQLString,
} = require('graphql');

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() {
          return (new Date).toLocaleString();
        },
      },
    },
  })
});

const query = `query { serverTime }`;

graphql(schema, query).then(result =>{
  console.log(result.data);
});

// Prints:
// {serverTime: "9/5/2016, 6:28:46 PM"}
```

```python
import graphene
from datetime import datetime


class Query(graphene.ObjectType):
  server_time = graphene.Field(graphene.String)

  def resolve_server_time(obj, args, context, info):
    return str(datetime.now())

schema = graphene.Schema(query=Query)


query = 'query { serverTime }'

result = schema.execute(query)
print(result.data)

#: Prints: OrderedDict([('serverTime', '2017-07-24
19:10:38.127089')])
```

Appier

Runnable GraphQL server code / Query Execution (query + schema → result)

```javascript
const {
  graphql, GraphQLSchema, GraphQLObjectType,
  GraphQLString,
} = require('graphql');

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() {
          return (new Date).toLocaleString();
        },
      },
    },
  })
});

const query = `query { serverTime }`;

graphql(schema, query).then(result =>{
  console.log(result.data);
});

// Prints:
// {serverTime: "9/5/2016, 6:28:46 PM"}
```

```python
import graphene
from datetime import datetime


class Query(graphene.ObjectType):
  server_time = graphene.Field(graphene.String)

  def resolve_server_time(obj, args, context, info):
    return str(datetime.now())

schema = graphene.Schema(query=Query)


query = 'query { serverTime }'

result = schema.execute(query)
print(result.data)

#: Prints: OrderedDict([('serverTime', '2017-07-24
19:10:38.127089')])
```

Appier

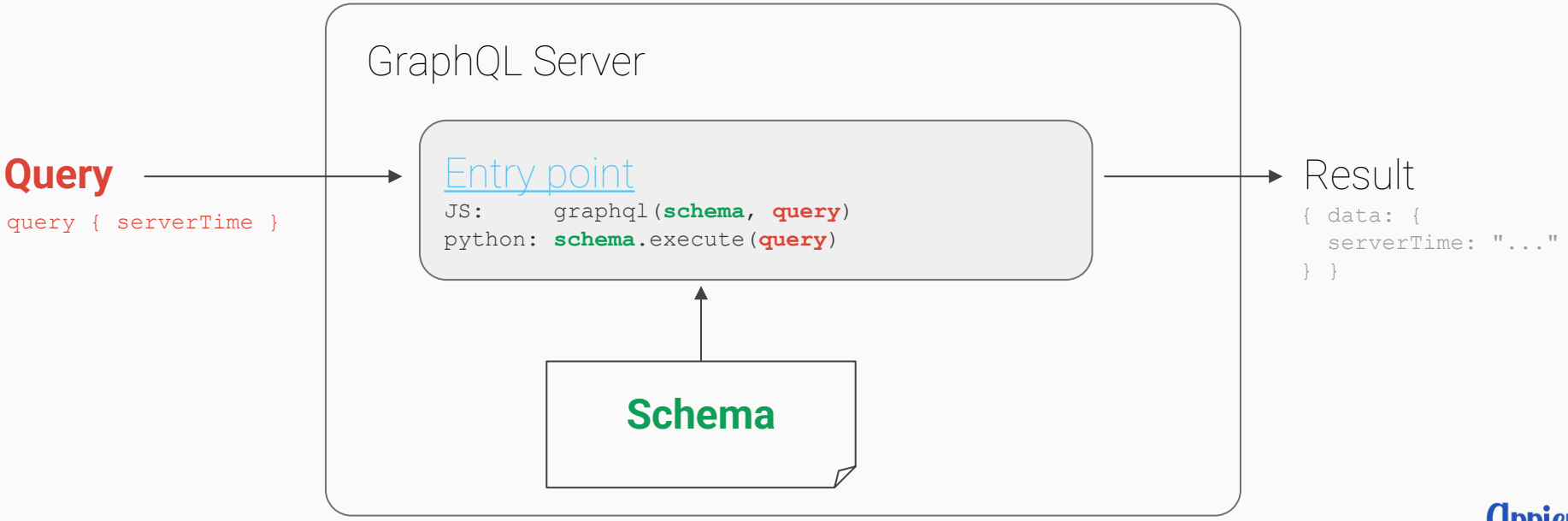
Runnable GraphQL server code / **Result**

```javascript
const {
  graphql, GraphQLSchema, GraphQLObjectType,
  GraphQLString,
} = require('graphql');

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() {
          return (new Date).toLocaleString();
        },
      },
    },
  })
});

const query = `query { serverTime }`;

graphql(schema, query).then(result =>{
  console.log(result.data);
});

// Prints:
// {serverTime: "9/5/2016, 6:28:46 PM"}
```

```python
import graphene
from datetime import datetime


class Query(graphene.ObjectType):
  server_time = graphene.Field(graphene.String)

  def resolve_server_time(obj, args, context, info):
    return str(datetime.now())

schema = graphene.Schema(query=Query)


query = 'query { serverTime }'

result = schema.execute(query)
print(result.data)

#: Prints: OrderedDict([('serverTime', '2017-07-24
19:10:38.127089')])
```

Appier

Parts in GraphQL server



Appier

GraphQL server over HTTP

```javascript
const {
  graphql, GraphQLSchema, GraphQLObjectType,
  GraphQLString,
} = require('graphql');

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() {
          return (new Date).toLocaleString();
        },
      },
    },
  })
});

const query = `query { serverTime }`;

graphql(schema, query).then(result =>{
  console.log(result.data);
});
```

```python
import graphene
from datetime import datetime


class Query(graphene.ObjectType):
  server_time = graphene.Field(graphene.String)

  def resolve_server_time(obj, args, context, info):
    return str(datetime.now())

schema = graphene.Schema(query=Query)


query = 'query { serverTime }'

result = schema.execute(query)
print(result.data)
```

Appier

GraphQL server over HTTP

```javascript
const {
  graphql, GraphQLSchema, GraphQLObjectType,
  GraphQLString,
} = require('graphql');

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() {
          return (new Date).toLocaleString();
        },
      },
    },
  })
});

app.post('/graphql', (req, res) => {
  graphql(schema, req.body).then(res.json);
});
```

```python
import graphene
from datetime import datetime


class Query(graphene.ObjectType):
  server_time = graphene.Field(graphene.String)

  def resolve_server_time(obj, args, context, info):
    return str(datetime.now())

schema = graphene.Schema(query=Query)


@app.route('/graphql')
def graphql():
  result = schema.execute(request.body)
  return json.dumps({
    data: result.data,
    errors: result.errors
  })
```

Appier

Ready-made GraphQL Server library

NodeJS

- express-graphql
- Apollo Server (real-world example)

Python

- Flask-GraphQL

Appier

Running Example

Clone

https://github.com/appier/graphql-cookbook

and follow the README

Appier

GraphiQL

localhost:5000/graphql?query

Johnson

localhost:5000/graphql

GraphiQL

Prettify

Docs

```
1 {
2   serverTime
3 }
```

```
{
  "data": {
    "serverTime": "2017-08-06 14:28:48.962753"
  }
}
```

Appier

GraphQL Results & error handling

Query

```
{
  serverTime
}
```

Result

```
{
  "data": {
    "serverTime":
    "9/5/2016, 6:28:46 PM"
  }
}
```

Runtime Error

```
{
  "data": {
    "serverTime": null,
  },
  "errors": {
    "message": "...",
    "locations": [...]
  }
}
```

Parse Error

```
{
  "errors": [
    {
      "message": "...",
      "locations": [...]
    }
  ]
}
```

Appier

Defining API shape - GraphQL schema

Appier

Query & Schema

```
query {
  serverTime
}
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() { /* ...*/ },
      },
    },
  })
})
```

```
class Query(graphene.ObjectType):
  server_time = graphene.Field(...)

  def resolve_server_time(obj, args,
context, info):
    # ...

schema = graphene.Schema(query=Query)
```

Appier

Object Type

```
query {
  serverTime
}
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() { /* ...*/ },
      },
    },
  })
})
```

```
class Query(graphene.ObjectType):
  server_time = graphene.Field(...)

  def resolve_server_time(obj, args,
context, info):
    # ...

schema = graphene.Schema(query=Query)
```

Appier

Object Type / **Fields**

```
query {
  serverTime
}
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTme: {
        type: GraphQLString,
        resolve() { /* ...*/ },
      },
    },
  })
})
```

```
class Query(graphene.ObjectType):
  server_time = graphene.Field(...)

  def resolve_server_time(obj, args,
context, info):
    # ...

schema = graphene.Schema(query=Query)
```

Appier

Object Type / Fields / **Resolvers**

```
query {
  serverTime
}
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        resolve() { /* ...*/ },
      },
    },
  })
})
```

```
class Query(graphene.ObjectType):
  server_time = graphene.Field(...)

  def resolve_server_time(obj, args,
context, info):
    # ...

schema = graphene.Schema(query=Query)
```

Appier

Object Type / Fields / **Types**

- Object Type
  - have "Fields"
  - each field have `type` & `resolve` function
- Scalar types
  - No more "fields", should resolve to a value
  - Built-in: String, Int, Float
  - Custom: specify how to serialize / deserialize. (NodeJS / Python)
  - Enum (NodeJS / Python) / server uses value, client uses name (scalar coercion)

Appier

Object Type / Fields / **Types (2)**

- Modifiers
  - Lists (NodeJS / Python)
  - Non-Null (NodeJS / Python)
- Interfaces
  - Defines fields what must be implemented in an object type
- Union Type
  - Resolves to a type at run time

Appier

Field with a Object Type

```
query {
  serverTime { hour, minute, second }
}
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: Time,
        resolve() { /* ...*/ },
      },
    },
  })
})
```

```
class Query(graphene.ObjectType):
  server_time = graphene.Field(Time)

  def resolve_server_time(obj, args,
context, info):
    # ...

schema = graphene.Schema(query=Query)
```

Appier

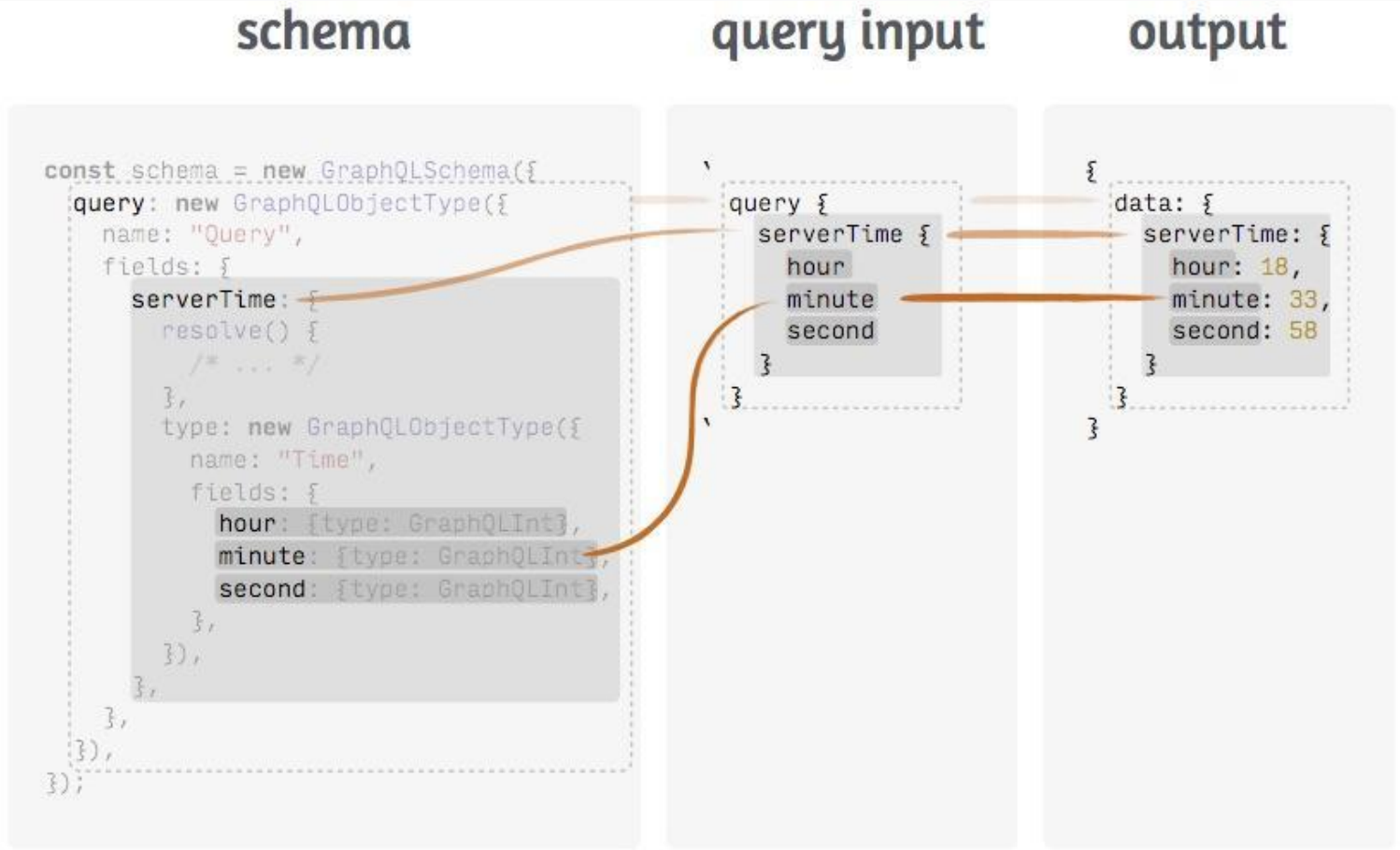
Field with a Object Type

```
query {
  serverTime { hour, minute, second }
}
```

```
const Time = new GraphQLObjectType({
  name: 'Time',
  fields: {
    hour: {type: GraphQLInt},
    minute: {type: GraphQLInt},
    second: {type: GraphQLInt},
  }
});
```

```
class Time(graphene.ObjeceType):
  hour = graphene.Int
  minute = graphene.Int
  second = graphene.Int
```

Appier



Schema, query and output

schema

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      article: {
        type: ArticleType,
        args: {...},
        resolve() {...}
      },
      articles: {
        type: new GraphQLList(ArticleType),
        resolve() {...}
      },
      reply: {
        type: new GraphQLObjectType({
          name: 'ReplyType',
          fields: {
            author: {
              type: userType,
              resolve() {...}
            },
            ...
          }
        }),
        resolve() {...}
```

query input

```
query {
  article(...) {...}
  reply {
    author {...}
  }
}
```

output

```
{
  data: {
    article: {...},
    reply: {
      author: {...}
    }
  }
}
```

Real-world example (simplified from Cofacts API server)

Resolving object fields

Appier

Resolving a field

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: Time,
        resolve() { /* ...*/ },
      },
    },
  })
});
```

```
class Query(graphene.ObjectType):
  serverTime = graphene.Field(Time)

  def resolve_server_time(obj, args,
context, info):
    # ...

schema = graphene.Schema(query=Query)
```

Appier

Resolver function signature

`resolve(obj, args, context)`

Appier

Resolver function signature / **obj**

`resolve(obj, args, context)`

- `obj`: The previous object, which for a field on the root Query type is often not used.

(Text from official documentation)

Appier

Resolver function signature / **obj**

```javascript
const Time = new GraphQLObjectType({
  name: "Time",
  fields: {
    hour: {
      type: GraphQLInt,
      resolve: obj => obj.getHours() },
    minute: {
      type: GraphQLInt,
      resolve: obj => obj.getMinutes() },
    second: {
      type: GraphQLInt,
      resolve: obj => obj.getSeconds() }, }
});

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: "Query",
    fields: {
      serverTime: {
        type: Time,
        resolve(){ return new Date; }
      }
    }
  })
});
```

```python
class Time(graphene.ObjectType):
  hour = graphene.Int()
  minute = graphene.Int()
  second = graphene.Int()

  def resolve_hour(obj, args, context, info):
    return obj.hour

  def resolve_minute(obj, args, context, info):
    return obj.minute

  def resolve_second(obj, args, context, info):
    return obj.second

class Query(graphene.ObjectType):
  server_time = graphene.Field(Time)

  def resolve_server_time(obj, args, context, info):
    return datetime.now()

schema = graphene.Schema(query=Query)
```

Appier

```
const query = `
  query {
    serverTime {
      hour
      minute
    }
  }
`;
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: "Query",
    fields: {
      serverTime: {
        type: Time,
        resolve(){
          const date = new Date;
          return date;
        }
      }
    }
  })
});
```

1.

```
const Time = new GraphQLObjectType({
  name: "Time",
  fields: {
    hour: {
      type: GraphQLInt,
      resolve(obj) {
        return source.getHours();
      }
    },
    minute: {
      type: GraphQLInt,
      resolve(obj) {
        return source.getMinutes();
      }
    },
    second: {
      type: GraphQLInt,
      resolve(source) {
        return source.getSeconds();
      }
    },
  }
});
```

2.

2.

order not guaranteed

Field selected in query, `resolve()` invoked

Field not selected, `resolve()` not invoked

How resolver functions are invoked

Trivial resolvers

```
someField(obj) {
  return obj.someField;
}
```

```
resolve_some_field(obj):
  return obj.some_field
```

```
const Time = new GraphQLObjectType({
  name: "Time",
  fields: {
    hour: {type: GraphQLInt},
    minute: {type: GraphQLInt},
    second: {type: GraphQLInt}});

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: "Query",
    fields: {
      serverTime: {
        type: Time,
        resolve(){
          const date = new Date;
          return {
            hour: date.getHours(),
            minute: date.getMinutes(),
            second: date.getSeconds(),
          };
        }
      }
```

```
class Time(graphene.ObjectType):
  hour = graphene.Int()
  minute = graphene.Int()
  second = graphene.Int()


class Query(graphene.ObjectType):
  server_time = graphene.Field(Time)

  def resolve_server_time(obj, args, context, info):
    return datetime.now()

schema = graphene.Schema(query=Query)
```

has property `hour`, `minute`, `second`

Appier

Root value

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: "Query",
    fields: {
      serverTime: {
        type: Time,
        resolve(obj){ ... }
      }
    }
  })
});

graphql(schema, query, rootValue)
```

```
class Query(graphene.ObjectType):
  server_time = graphene.Field(Time)

  def resolve_server_time(obj):
    return ...



schema = graphene.Schema(query=Query)
schema.execute(
  query,
  root_value=root_value
)
```

Appier

Resolver function signature / **args**

`resolve(obj, **args**, context)`

- `obj`: The previous object, which for a field on the root Query type is often not used.
- `args`: The arguments provided to the field in the GraphQL query.

(Text from official documentation)

Appier

Arguments when querying a field

```
query {
  serverTime(timezone: "UTC") {hour}
}
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      serverTime: {
        type: GraphQLString,
        args: {
          timezone: {
            type: GraphQLString,
            description: 'Timezone name (Area/City)',
          },
        },
        resolve(obj, { timezone = 'Asia/Taipei' }) {
          return (new Date()).toLocaleString({
            timeZone: timezone,
          });;
        },
      },
    },
  }),
});
```

```
query {
  serverTime(timezone: 0) {hour}
}
```

```
class Query(graphene.ObjectType):
  server_time = graphene.Field(
    graphene.String,
    timezone=graphene.Argument(
      graphene.Int,
      default_value=8,
      description="UTC+N, N=-24~24."
    )
  )


  def resolve_server_time(obj, args, context, info):
    tz = timezone(timedelta(hours=args['timezone']))
    return str(datetime.now(tz))


schema = graphene.Schema(query=Query)
```

Appier

Args and Input Object Type

```
query {
  serverTime ( timezone: "UTC", offset: { hour: 3, minutes: 30 } ) {
    hour
  }
}
```

- Input Object types are Object types without arguments & resolvers
- Example (NodeJS, Python)
- Extended reading: filters, sorting, pagination, ...

Appier

Resolver function signature / **context**

`resolve(obj, args, context)`

- `obj`: The previous object, which for a field on the root Query type is often not used.
- `args`: The arguments provided to the field in the GraphQL query.
- `context`: A value which is provided to every resolver and holds important contextual information like the currently logged in user, or access to a database.

(Text from official documentation)

Appier

Context

```
// Time object type's field
hour: {
  type: GraphQLInt,
  resolve(obj, args, context) {...}
}

// Root Query type's field
serverTime: {
  type: Time,
  resolve(obj, args, context) {...},
}

// Executing query
graphql(
  schema, query, rootValue, context
)
```

```
#: Time object's hour resolver
def resolve_hour(obj, args, context, info):
  return ...


#: Root Qyery type's serverTime resolver
def resolve_server_time(
  obj, args, context, info
):
  return ...

#: Execute the query
schema = graphene.Schema(
  query=Query, context=context)
```

Appier

Mutative APIs

```
query {
  createUser(name:"John Doe"){
    id
  }
}
```

Appier

```
query {
  user(name:"John Doe"){ id }
  article(id:123) { text }
}
```

Run in parallel

```
mutation {
  createUser(...) {...}
  createArticle(...) {...}
}
```

Run in series

Appier

Mutations

```javascript
const CreatePersonResult = new GraphQLObjectType({
  name: 'CreatePersonResult', fields: {
    ok: { type: GraphQLBoolean },
    person: { type: Person },
  }
})

const schema = new GraphQLSchema({
  query: ...,
  mutation: new GraphQLObjectType({
    name: 'Mutation',
    fields: {
      CreatePerson: {
        type: CreatePersonResult,
        args: { name: { type: GraphQLString } },
        resolve(obj, args) {
          const person = {name: args.name};
          // Should do something that persists
          // the person
          return { ok: true, person };
        }
      }
    }
  })
})
```

```python
class CreatePerson(graphene.Mutation):
  class Input:
    name = graphene.Argument(graphene.String)

  ok = graphene.Field(graphene.Boolean)
  person = graphene.Field(lambda: Person)

  @staticmethod
  def mutate(root, args, context, info):
    #: Should do something that persists
    #: the new Person here
    person = Person(name=args.get('name'))
    ok = True
    return CreatePerson(person=person, ok=ok)

class Mutation(graphene.ObjectType):
    create_person = CreatePerson.Field()

schema = graphene.Schema(
  query=..., mutation=Mutation
)
```

Appier

Mutations

```javascript
const CreatePersonResult = new GraphQLObjectType({
  name: 'CreatePersonResult', fields: {
    ok: { type: GraphQLBoolean },
    person: { type: Person },
  }
})

const schema = new GraphQLSchema({
  query: ...,
  mutation: new GraphQLObjectType({
    name: 'Mutation',
    fields: {
      CreatePerson: {
        type: CreatePersonResult,
        args: { name: { type: GraphQLString } },
        resolve(obj, args) {
          const person = {name: args.name};
          // Should do something that persists
          // the person
          return { ok: true, person };
        }
      }
    }
  })
})
```

```python
class CreatePerson(graphene.Mutation):
  class Input:
    name = graphene.Argument(graphene.String)

  ok = graphene.Field(graphene.Boolean)
  person = graphene.Field(lambda: Person)

  @staticmethod
  def mutate(root, args, context, info):
    #: Should do something that persists
    #: the new Person here
    person = Person(name=args.get('name'))
    ok = True
    return CreatePerson(person=person, ok=ok)

class Mutation(graphene.ObjectType):
    create_person = CreatePerson.Field()

schema = graphene.Schema(
  query=..., mutation=Mutation
)
```

Appier

Mutations

```
const CreatePersonResult = new GraphQLObjectType({
  name: 'CreatePersonResult', fields: {
    ok: { type: GraphQLBoolean },
    person: { type: Person },
  }
})

const schema = new GraphQLSchema({
  query: ...,
  mutation: new GraphQLObjectType({
    name: 'Mutation',
    fields: {
      CreatePerson: {
        type: CreatePersonResult,
        args: { name: { type: GraphQLString } },
        resolve(obj, args) {
          const person = {name: args.name};
          // Should do something that persists
          // the person
          return { ok: true, person };
        }
      }
    }
  })
})
```

```
class CreatePerson(graphene.Mutation):
  class Input:
    name = graphene.Argument(graphene.String)

  ok = graphene.Field(graphene.Boolean)
  person = graphene.Field(lambda: Person)

  @staticmethod
  def mutate(root, args, context, info):
    #: Should do something that persists
    #: the new Person here
    person = Person(name=args.get('name'))
    ok = True
    return CreatePerson(person=person, ok=ok)

class Mutation(graphene.ObjectType):
    create_person = CreatePerson.Field()

schema = graphene.Schema(
  query=..., mutation=Mutation
)
```

Appier

Extended reading -- Designing GraphQL Mutations

Designing GraphQL Mutations

Johnson

安全 | https://dev-blog.apollodata.com/designing-graphql-mutations-e09de826ed97

應用程式 JIRA 通訊錄 Regexper jenkins Dev / Staging / Pro... Rollbar Rollbar POST

Designing GraphQL Mutations

Design principles for building effective GraphQL mutation systems that can evolve over time

Designing a good GraphQL API is tricky, because you always want to balance utility and convenience with a consideration around how the API may evolve in the future.

The main points to consider when designing your GraphQL mutations are:

- **Naming.** Name your mutations verb first. Then the object, or “noun,” if

Appier

Making requests to GraphQL Servers

Appier

Talk to GraphQL server via HTTP

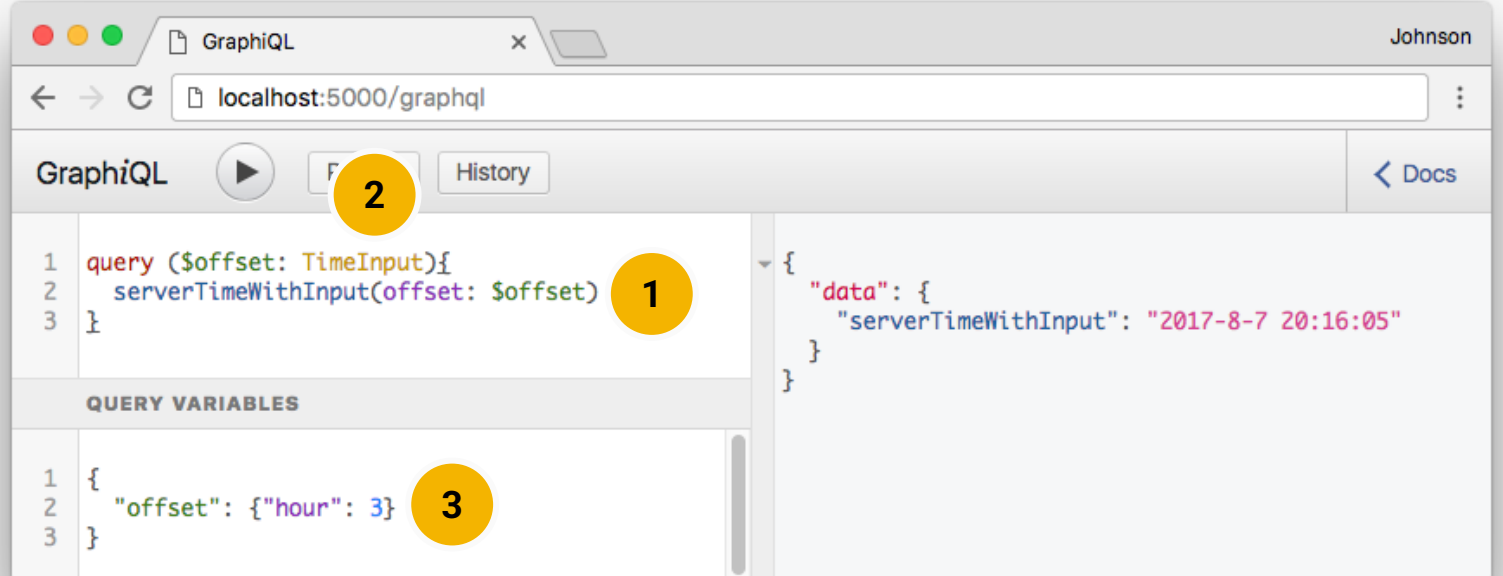
- Depends on server (apollo-server / Flask-GraphQL) implementation
- Inspect graphiql network requests
- Mostly supports:

```
POST /graphql HTTP/1.1
Content-Type: application/json

{
  "query": "...GraphQL Query string...",
}
```

Appier

Working with GraphQL Variables (Text from official documentation)

1. Replace the static value in the query with `$variableName`
2. Declare `$variableName` as one of the variables accepted by the query
3. Pass `variableName: value` in the separate, transport-specific (usually JSON) variables dictionary



Appier

Working with GraphQL Variables (Text from official documentation)

1. Replace the static value in the query with `$variableName`
2. Declare `$variableName` as one of the variables accepted by the query
3. Pass `variableName: value` in the separate, transport-specific (usually JSON) variables dictionary

```
POST /graphql HTTP/1.1
Content-Type: application/json

{
  "query":
    "query($offset:TimeInput){serverTimeWithInput(offset:$offset)}",
  "variables": {
    "offset": { "hour": 3 }
  }
}
```

Some old GraphQL servers only accept "`variables`" as JSON **strings**

Appier

Clients

- graphiql (for development)
- curl
- XMLHttpRequest / fetch
  - Example
- Client library - apollo-client
  - HOC (can be configured to use custom redux store)
  - Store normalization
  - Instance-level caching
  - Pre-fetching

Appier

Advanced query techniques

- Field alias - rename object key in result
- Fragments - to dedup fields
- Inline Fragments - for union types
- Directives - selectively query for fields

Extended reading: The Anatomy of GraphQL queries

Appier

Solving N+1 Problem: Dataloader

Appier

```
const query = `
  query {
    serverTime {
      hour
      minute
    }
  }
`;
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: "Query",
    fields: {
      serverTime: {
        type: Time,
        resolve(){
          const date = new Date;
          return date;
        }
      }
    }
  })
});
```

1.

```
const Time = new GraphQLObjectType({
  name: "Time",
  fields: {
    hour: {
      type: GraphQLInt,
      resolve(obj) {
        return source.getHours();
      }
    },
    minute: {
      type: GraphQLInt,
      resolve(obj) {
        return source.getMinutes();
      }
    },
    second: {
      type: GraphQLInt,
      resolve(source) {
        return source.getSeconds();
      }
    },
  }
});
```

2.

2.

order not guaranteed

Field selected in query, `resolve()` invoked

Field not selected, `resolve()` not invoked

How resolver functions are invoked

N+1 problem

```
{
  users {
    bestFriend {
      displayName
    }
  }
}
```

1. Resolver for `users` field queries database, returns a list of `N` users -- 1 query
2. For each user, resolver for `bestFriend` field is fired
3. For each call to `bestFriend`'s resolver, it queries database with `bestFriendId` to get the user document -- N queries

Appier

Tackling N+1 problem (1)

```
{
  users {
    bestFriend {
      displayName
    }
  }
}
```

Solution 1: Resolve `bestFriend` in `users`' resolver

- Couples different resolving logic
- Need to know if "`bestFriend`" is queried

- Not recommended

Appier

Tackling N+1 problem (2)

```
{
  users {
    bestFriend {
      displayName
    }
  }
}
```

Solution 2: Query all `bestFriend` in a batch

- After `users`' resolver return, `bestFriend`'s resolver is fired N times synchronously.
- Collect all `bestFriendIds` and query the database at once
- With the help of dataloader

Appier

Dataloader - Batching & caching utility

```
import DataLoader from 'dataloader';

const userLoader = new DataLoader(
  ids => {
    console.log('Invoked with', ids);
    return User.getAllByIds(ids);
  }
);
```

```
from aiodataloader import DataLoader

class UserLoader(DataLoader):
  async def batch_load_fn(self, ids):
    print('Invoked with %s' % ids)
    return User.get_all_by_ids(ids)

user_loader = UserLoader()
```

Define a batch function

```
userLoader.load(1).then(console.log)
userLoader.load(2).then(console.log)
userLoader.load(3).then(console.log)
userLoader.load(1).then(console.log)
```

```
future1 = user_loader.load(1)
future2 = user_loader.load(2)
future3 = user_loader.load(3)
future4 = user_loader.load(1) # == future1
```

Call load() whenever you want to

```
// Outputs:
// Invoked with [1,2,3]
// {id: 1, ...}
// {id: 2, ...}
// {id: 3, ...}
// {id: 1, ...}
```

```
# prints:
# Invoked with [1, 2, 3]
print(await future1) # {id: 1, ...}
print(await future2) # {id: 2, ...}
print(await future3) # {id: 3, ...}
print(await future4) # {id: 1, ...}
```

Get data you need

Appier

Batch function

```
const userLoader = new DataLoader(ids => {
  // Fake data loading
  return Promise.resolve(ids.map(id => ({id})))
})
```

```
class UserLoader(DataLoader):
  async def batch_load_fn(self, ids):
    """Fake data loading"""
    return [{'id': id} for id in ids]
```

- **Maps** N IDs to N documents
- Input: IDs
  - Output are cached by ID
- Output: `Promise<documents>`
- Length of output must match input
  - If not found, return null
  - *i*-th ID in input should match *i*-th document in output

Appier

dataloader instance methods

- `load(id)`:
  - input: 1 ID
  - output: a promise that resolves to 1 loaded document
- `loadMany(ids)`:
  - input: N IDs
  - output: a promise that resolves to N loaded documents

Appier

Multiple dataloader instances

- Prepare a batch function for each data-loading mechanism
- Create a dataloader instance for each batch function
- Batching & caching based on instances

```javascript
const userLoader = new DataLoader(...)
const articlesByAuthorIdLoader = new DataLoader(...)

// Get user 1's best friends's articles
userLoader.load(1).then(
  ({bestFriendId}) =>
    articlesByAuthorIdLoader.load(bestFriendId)
).then(console.log)


// prints:
// [article1, article2, ...]
```

```python
user_loader = UserLoader()
articles_by_author_id_loader = ArticlesByAuthorIdLoader()

# Get user 1's best friends's articles
user = await user_loader.load(1)
print(await articles_by_author_id_loader.load(
  user.best_friend_id
))


# prints:
# [article1, article2, ...]
```

Appier

Combine dataloader with GraphQL schema

```
app.post('/graphql', bodyParser.json(),
  graphqlExpress(req => ({
    schema,
    context: {
      userLoader: new DataLoader(...),
      articleByUserIdLoader: new DataLoader(...)
    }
  }))
)
```

```
// field "user" in root query type
{ type: User,
  resolve(obj, {id}, {userLoader}) {
    return userLoader.load(id);
  } }

// field "bestFriendArticles" in User object type
{ type: new GraphQLList(Article),
  resolve({bestFriendId}, args,
    {articleByUserIdLoader}
  ) {
    return articleByUserIdLoader.load(bestFriendId);
  } }
```

```
class ViewWithContext(GraphQLView):
  def get_context(self, request):
    return {
      'user_loader': UserLoader(),
      'articles_by_author_id_loader':
        ArticlesByAuthorIdLoader()
    }
app.add_url_rule(
  '/graphql', view_func=ViewWithContext.as_view(
    'graphql', schema=schema, graphiql=True,
    executor=AsyncioExecutor) )
```

```
# in root query type
user = graphene.Field(User)
def resolve_user(obj, args, context):
  return context['user_loader'].load(args['id'])


# in user object type
best_friend_articles =
  graphene.Field(graphene.List(Article))
def resolve_best_friend_articles(obj, args, context):
  return context['articles_by_author_id_loader'].load(
    obj['best_friend_id']
  )
```

Appier

Summary

- RESTful endpoints --> GraphQL schema fields
- Strongly typed input / output
- Validation & Documentation

Other Resources

- "Extended reading" in this slide
- GraphQL official doc
- How to GraphQL online course
- (JS only) generate schema from GraphQL schema language
- Case studies
- FB group - GraphQL Taiwan