GraphQL

A query language for your API

JWORKS

POWERED BY ORDINA

Hi, my name is Sergei Storozhenko

Java Developer @Ordina Belgium

Cloud Automation Engineer @TVH

Why REST can be “problematic”?

- REST is an **architectural concept** for network-distributed software first presented in PhD of Roy Fielding in 2000
- It has no official set of tools
- It has no specification
- It can use HTTP but also any other protocol
- It does not cared about the payload (JSON, XML, ….)
- Main purpose of it to decouple an API from the client.

”Problems" of REST

- Not adhering to the specs and too wide interpretation of good practices
- “Problems” arising from the nature of REST

Not adhering to specs

- Currently, the “default” way of implementing REST is via HTTP with JSON as a payload
- Thus, it should adhere to HTTP specs like using correct HTTP methods, headers, statuses, etc.
- There is no mechanism to enforce the adherence to these specs
- Rather broad interpretation of "good practices". Everybody creates own “good practices”
- Wrong usage of HTTP semantics (methods, URL structure, status codes, etc.)
- Decent HATEOAS implementation adds a lot of technical overhead -> a lot of teams opt for not using it at all

Example: Jenkins Role Strategy Plugin API

- GET /role-strategy/strategy/getAllRoles
- POST /role-strategy/strategy/addRole
- POST / role-strategy/strategy/assignRole
- POST / role-strategy/strategy/unassignRole

Schemaless: problems

- Heavy reliance on the up-to-day documentation as the only mean to define a contract between server and client. There is JSON Scheme specification but I have never seen it being used.
- While REST is claimed to be ”designed for evolution”, in fact, the evolution of API and versioning is not straightforward. Ideally, it should evolve without versions (non-breaking changes, tolerant reader)
- Need in consumer contract tests to assure no breaking changes are introduced

Underfetching

- To get all necessary info a client has to navigate through a number endpoints. This is actually the essence of REST but may be a disadvantage in some use cases (n +1 request problem), where performance is important
- Typical solution is embedding resources but this can create a problem of overfetching for other clients

Overfetching

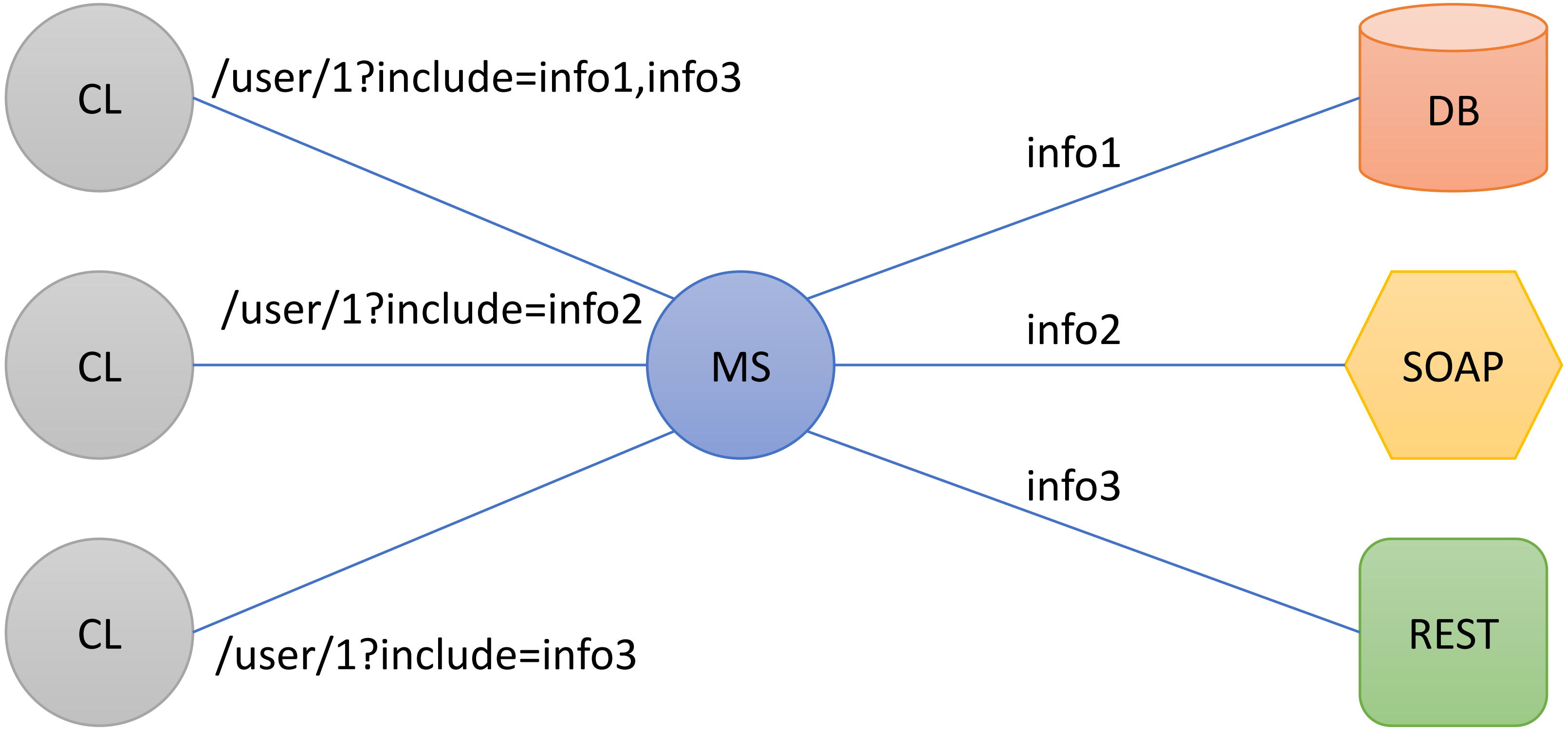
Needed

```
{
  "firstName": "John",
  "lastName": "Doe",
  "mobile": "047856389076"
}
```

Received

```
{
  "firstName": "John",
  "lastName": "Doe",
  "mobile": "047856389076",
  "email": john.doe@ordina.be,
  "businessUnit": "jwork”,
  ”function": ”java_consultant”,
   “car”: {
       ”make”: “BMW”
        . . . . . . .
    }
}
```

Using partials



GraphQL to the rescue?

- GraphQL was developed internally by Facebook in 2012
- Open sourced in 2015
- List of GraphQL users is growing fast

f
intuit®
coursera
S

What is GraphQL?

- It is a query language
- It is also a specification
- It has a set of specific tools
- It operates via a single endpoint
- Optimized for flexibility and performance

Get exactly what you want

Query

```
{
  getEmployee(id: 1){
    firstName
    lastName
  }
}
```

Response

```
{
  ”data": {
    "getEmployee": {
      "firstName": "John",
      "lastName": "Doe"
    }
  }
}
```

Get exactly what you want

Query

```
{
  getEmployee(id: 1){
    firstName
    lastName
    email
    mobile
  }
}
```

Response

```
{
  ”data": {
    "getEmployee": {
      "firstName": "John",
      "lastName": "Doe",
      "email": ”john.doe@tvh.com",
      "mobile": ”0478453678"
    }
  }
}
```

Most commonly operates via HTTP and single endpoint

- `GET http://myapi/graphql?query={me{name}}`
- `POST http://myapi/graphql`
  with body:

```
{
   “query”: “{me{name}}”
}
```

GraphQL is strongly typed

Object types

```
type Employee {
   id: Int!
   firstName: String!
   lastName: String!
   email: String!
   mobile: String!     #non-nullable
   car: Car            #nullable
}
```

Scalars

- **Int**: A signed 32-bit integer.
- **Float**: A signed double-precision floating-point value.
- **String**: A UTF-8 character sequence.
- **Boolean**: true or false.
- **ID**: The ID scalar type represents a unique identifier, often used to re-fetch an object or as the key for a cache. The ID type is serialized in the same way as a String; however, defining it as an ID signifies that it is not intended to be human-readable.

Other types

- Enums
- Interfaces
- Union type
- Input type
- GraphQL has also variables that can be used as query arguments
- Fragments that can be re-used in different queries

The whole API is defined in scheme

- Scheme defines all possible types and actions
- Scheme is an actual contract for all service – client interactions
- Scheme has a tree structure, hence, the name GraphQL

Special root types (actions)

```
schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
 }
```

Query is idempotent (read)

```
type Query {
  employees: [Employee]
  employee(id: Int): Employee
}
```

```
type Employee {
   id: Int!
   firstName: String!
   lastName: String!
   email: String!
   mobile: String!
}
```

Mutation is used to change state (create, update, delete)

```
type Mutation {

  addEmployee(firstName: String!, lastName: String!,
              email: String!,
              mobile: String!): Employee

  updateEmployee(id:Int, firstName: String!, lastName: String!,
              email: String!,
              mobile: String!): Employee
}
```

Add employee example

```
mutation {
  addEmployee(firstName: “John”, lastName: “Doe”,
    email: “john.doe@tvh.com”, mobile: “0478674598”){
         id
         firstName
         lastName
         email
         mobile
     }
}
```

Subscription is used to follow real time changes

```
type Subscription {
  stockQuotes(stockCodes:[String]) : StockPriceUpdate!
}

type StockPriceUpdate {
  dateTime : String
  stockCode : String
  stockPrice : Float
  stockPriceChange : Float
}
```

Example: https://github.com/graphql-java/graphql-java-subscription-example

Introspection: the whole schema can be examined by the client

__Schema

__Type

__TypeKind

__Field

__InputValue

__EnumValue

__Directive

Queries can be easily validated against the scheme before the execution

```
{
  getEmployee(id: 1) {
    firstName
    lastName
    email
    mobile
    address
  }
}
```

```
{
  "data": null,
  "errors": [
    {
      "message": "Validation error of type FieldUndefined: Field 'address' in type 'Employee' is
undefined @ 'getEmployee/address'",
      "locations": [
        {
          "line": 7,
          "column": 5
        }
      ],
      "description": "Field 'address' in type 'Employee' is undefined",
      "validationErrorType": "FieldUndefined",
      "queryPath": [
        "getEmployee",
        "address"
      ],
      "errorType": "ValidationError",
      "path": null,
      "extensions": null
    }
  ]
}
```

Query execution

- We can think of each field in a GraphQL query as a function or method of the previous type which returns the next type.
- Each field on each type is backed by a function called the ***resolver*** which is provided by the GraphQL server developer.
- When a field is executed, the corresponding ***resolver*** is called to produce the next value.
- If a field produces a scalar value like a string or number, then the execution completes.

Query execution

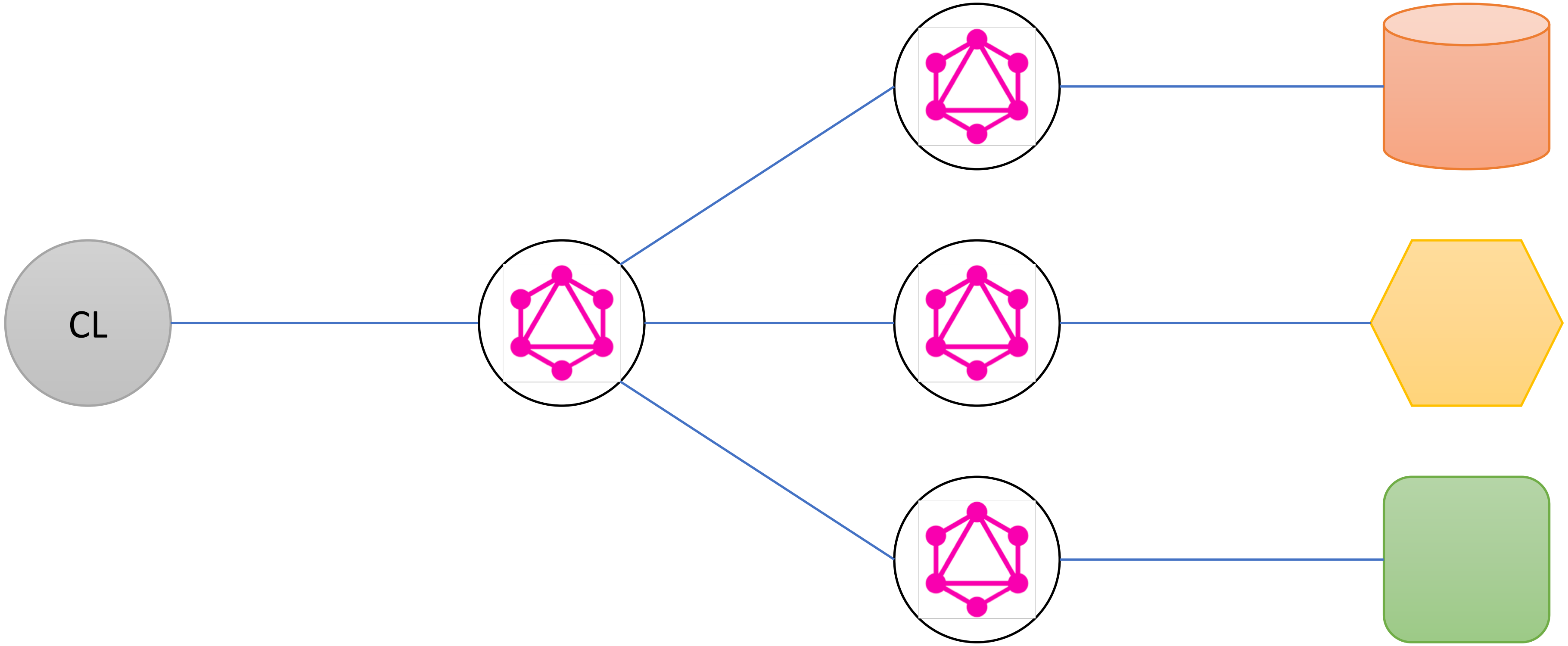
```
{
  getEmployee(id: 1) {
    email
    car {
      make
      model
      leasingCompany {
        name
        contactPerson
        phone
    }
  }
}
```

```
Employee getEmployee(int id){
   return repo.getEmployee(1);
}
```

```
Car getCar (Employee empl){
 return repo.findCar(empl.getLicencePlate);
}
```

```
LeasingCompany getCompany (Car car){
 return repo.findCompany(car.getCompanyName);
}
```

Schema stitching



Apollo client for Andriod: https://github.com/apollographql/apollo-android

Evolve API without versions

```
type Employee {
   id: Int!
   firstName: String!
   lastName: String!
   email: String!
   mobile: String!
   tel: String @deprecated
 + socialSecurityNr: String
   car: Car
}
```

Intelligent field deprecation

- By analysis of all executed queries, GraphQL server can figure out when deprecated fields are not in use anymore. After this the deprecated fields can safely be removed.

Solving n + 1 request problem in GraphQL

- In GraphQL n + 1 request problem still exists but it is shifted from the client side to the server side.
- Basically there are a couple of solutions:
  - Asynchronous calls
  - Data loaders

```
{
  getEmployees {
    email
    car {
      make
      model
    }
  }
}
```

Retrieving a list of 10 employees will result in 11 queries:

1 query to get the list of employees

10 queries to get a car for each employee from the list.

Asynchronous resolvers

```
CompletableFuture<Car> car(Employee empl){
    return CompletableFuture.supplyAsync(
        () -> carClient.findCarByLicensePlate(
        empl.getCarLicencePlate());
}
```

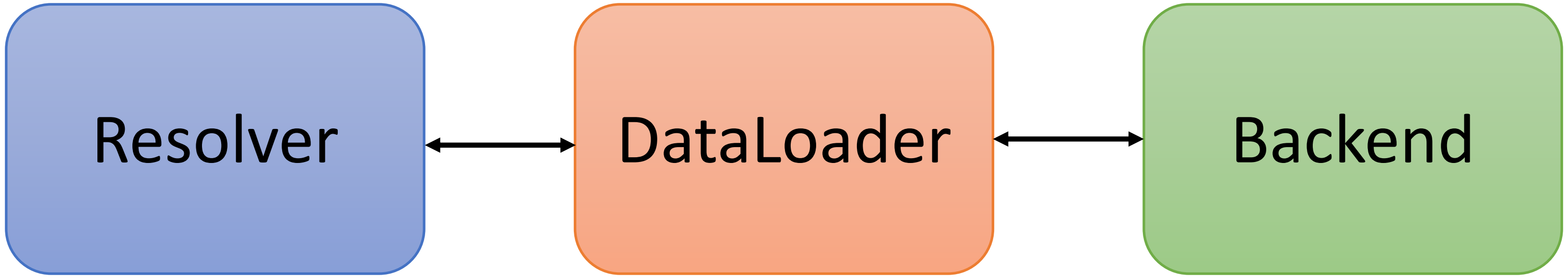
Asynchronous resolvers

- Would increase performance by making concurrent calls instead of consecutive
- Would work efficiently, if the concurrent calls are made to different backends (APIs, DBs, etc.)
- In case of calls to the same backend performance will be limited by the concurrent call handling capabilities of the particular backend.

DataLoader

DataLoader is a generic utility to be used as part of your application's data fetching layer to provide a simplified and consistent API over various remote data sources such as databases or web services via **batching and caching**.

DataLoader



DataLoader: batching

```
{
  getEmployees {
    email
    car {
      make
      model
    }
  }
}
```

getEmpoloyees()

~~getCar(1)~~

~~getCar(2)~~

~~. . . . .~~

~~getCar(n)~~

getCars(1,2,. . . n)

Dataloader: caching

```
{
  getEmployees {
    email
    car {
      make
      model
      leasingCompany {
        name
        contactPerson
        phone
    }
  }
}
```

Results of the repeated calls like

`getLeasingCompany(Employee empl)`

will be cached in the **scope of one request**

Caching

- In an endpoint-based API (like REST), simple HTTP caching can be used.
- The URL in these APIs is a **globally unique identifier** that the client can leverage to build a cache.
- In GraphQL, though, there's no URL-like primitive that provides this globally unique identifier for a given object. **It's hence a best practice for the API to expose such an identifier for clients to use**.
- One of the possibilities is providing object IDs as a globally unique identifiers

Caching

```
{
  getEmployee(id: “6a96e844fa8e33670b41f408ed83a245923af754”)
  }
    firstName
    lastName
    email
    mobile
    address
  }
}
```

Blocking malicious queries

- Size limiting
- Depth limiting
- Query whitelisting
- Query cost analysis

Implementing GraphQL

JS reference implementation:

https://graphql.org/graphql-js/

Apollo GraphQL platform

https://www.apollographql.com/

Implementing GraphQL

JS reference implementation:

https://graphql.org/graphql-js/

Apollo GraphQL platform

https://www.apollographql.com/

Implementing GraphQL in Java

Official GraphQL implementation:

https://github.com/graphql-java/graphql-java

Good place to start (schema-first approach):

https://www.graphql-java-kickstart.com/

Code-first approach library:

https://github.com/leangen/graphql-spqr

GraphQL client for Android:

https://github.com/apollographql/apollo-android

An in-browser IDE for exploring GraphQL

https://github.com/graphql/graphiql

When would you use GraphQL?

- If you need a highly query-able API
- If you expect an array of clients that need small and different data
- If you can restructure your data to be inexpensive to query
- **Then GraphQL is likely to fit your needs**.

And what about REST?

- If it allows careful evolution instead of global versioning
- If it serializes data instead of returning directly from data store
- If it implements sparse fieldsets to allow slimming down response sizes
- If it GZips contents
- If it outlines data structures with JSON Schema
- If it follows other know good practices
- **then the advertised advantages of GraphQL seem to fall a bit short**.

From Fil Strurgeon blog

Where to learn more?

- The place to start is the official GraphQL website: https://graphql.org/
- Further, use Google. There are plenty of tutorials, code examples, blogs, videos, etc.

Hands-on

Checkout the repo and follow the instructions

https://github.com/ordina-jworks/graphql-workshop