**REST or GraphQL? A Performance Comparative Study(Migrate to GraphQL)**

GraphQL allows clients to query a database represented by a schema. Such a design choice represents a significant paradigm shift from REST services: in REST, servers implement a list of access points (resources) that can be requested by clients; in contrast, GraphQL services map a database as a data graph, which can be queried by clients. Thus, the complexity of requesting data is shifted from the server to the client.

QP1: What is the average number of requests, in seconds, per architectural model of web service?

QP1 aims to analyze the average number of requests per second that the system is capable of handling.

QP2: What is the average time per request, in milliseconds, per architectural model of web service?

QP2 analyzes the average time, in milliseconds, for a single request.

QP3: What is the average time per concurrent request, in milliseconds, per architectural model of web service?

QP3, on the other hand, aims to measure the average time, in milliseconds, for requests where there is concurrency

QP4: What is the average transfer rate of documents, in KBytes per second, per architectural model of web service?

QP4 seeks to answer what the transfer rate of the responses sent by the services is, in KBytes per second.

latency and volume. That is, the delivery time (latency) and the transfer rate of the documents (volume) delivered to the client.

Implementation and Migration Process:

During the migration study, the target applications were refactored from the REST architectural model to GraphQL. This refactoring involved mapping (endpoints) typical of the REST architecture to types in GraphQL.

During the migration of the applications, it was also possible to observe that refactoring REST services to use **nested queries** using GraphQL is not a trivial（繁琐） task. For each existing resource exposed as a point of access in a REST service, a type is required in GraphQL. A mapping needs to be done, which is not always a simple operation because often there are relationships between such resources, resulting in a GraphQL schema that will generate highly nested queries. Therefore, refactoring such systems to serve a large graph data structure can quickly become a complex task of reengineering depending on the size of the system being migrated.

Conclusion:

The primary focus was on the performance of services in terms of requests per second, response time (latency), and data volume.

GraphQL is not a solution to eliminate all problems associated with API implementation. Performance bottlenecks can occur, and the main causes can be difficult to identify. Nevertheless, it may be less costly to maintain and evolve applications and APIs in GraphQL.

**REST vs GraphQL: A Controlled Experiment**

In REST-based APIs, data is exposed by means of *endpoints*.

An example of an endpoint is:  
GET /users/torvalds/repos

This endpoint returns the list of public repositories of a given user, e.g., *torvalds*.

The following listing shows a fragment of the returned JSON. full name (line 3), owner (line 5–8), created at (line 10),

among others.

1 [

2 {

3 "full\_name": "torvalds/libdc-for-dirk",

4 "private": false,

5 "owner": {

6 "login": "torvalds",

7 ...

8 },

9 "created\_at": "2017-01-17T00:25:49Z",

10 ...

11 },

12 ...

13 ]

In GraphQL, service data is exposed as a graph .

Each node of this graph/schema represents objects and contains fields. Each field has a name and a type. Edges appear when a field references another object. Clients access a GraphQL service through a single endpoint, which is used to submit queries.

文本

描述已自动生成

In GraphQL schemas, queries are defined using a special type, called Query. The following listing shows a fragment of a Query type, with only one query, called repository

(line 3), which has two parameters: name and owner. Both parameters are of type String. This query returns an object of a Repository type.

图片包含 文本

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

文本

描述已自动生成

Interested part,

Implementation time between two APIs

1. different types of queries
2. for graduate and undergraduate students

1. During the migration study, the target applications were refactored from the REST architectural model to GraphQL. This refactoring involved mapping (endpoints) typical of the REST architecture to types in GraphQL. Refactoring REST services to use **nested queries** using GraphQL is not a trivial task.

1. A requirement for migration was the coexistence of original architectural layers and REST interfaces together with an additional GraphQL interface. For this reason, GraphQL was integrated as part of the existing server. Only the API layer had to be expanded.
2. In order to implement the GraphQL specification, the open-source library graphql-java was used. The library provides three components that must be added to the existing system: the GraphQL execution unit, a schema and resolvers.
3. To achieve a homogeneous API the GraphQL schema was derived from the existing REST resources. In the GraphQL schema references between REST resources were mapped to references between GraphQL object types.

Integrating complex semi-compliant technologies in the context of distributed

systems naturally introduce challenges.

**Challenges on Architectural Level**

-Resource Identification through URIs.

-Uniform Interface.

-Self-Descriptive Messages

-Stateful Interactions through Hyperlinks.

**Challenges on Service-Level**

-Caching

-Denial of Service Attacks

curl --request POST \

--header 'content-type: application/json' \

--url 'https://swapi-graphql.netlify.app/.netlify/functions/index' \

--data '{"query":"query Query {\n allFilms {\n films {\n title\n director\n releaseDate\n speciesConnection {\n species {\n name\n classification\n homeworld {\n name\n }\n }\n }\n }\n }\n}"}'