FullStack.Cafe - Kill Your Tech Interview

Q1: Is GraphQL a Database Technology? ☆

Topics: GraphQL

Answer:

No. GraphQL is often confused with being a database technology. This is a misconception, GraphQL is a *query language* for APIs - not databases. In that sense it's database agnostic and can be used with any kind of database or even no database at all.

Q2: Is GraphQL only for React/JavaScript Developers? ☆

Topics: GraphQL

Answer:

No. GraphQL is an API technology so it can be used in any context where an API is required.

On the *backend*, a GraphQL server can be implemented in any programming language that can be used to build a web server. Next to Javascript, there are popular reference implementations for Ruby, Python, Scala, Java, Clojure, Go and .NET.

Since a GraphQL API is usually operated over HTTP, any client that can speak HTTP is able to query data from a GraphQL server.

Note: GraphQL is actually transport layer agnostic, so you could choose other protocols than HTTP to implement your server.

Q3: What is GraphQL? ☆

Topics: GraphQL

Answer:

GraphQL is a query language created by Facebook in 2012 which provides a **common interface between the client and the server for data fetching and manipulations**.

The client asks for various data from the GraphQL server via queries. The response format is described in the query and defined by the client instead of the server: they are called **client-specified queries**.
The structure of the data is not hardcoded as in traditional REST APIs - this makes retrieving data from the server more efficient for the client.

Q4: What is an *exclamation point* in GraphQL? ☆

Topics: GraphQL

Answer:

That means that the field is non-nullable. By default, all types in GraphQL are nullable. When non-null is applied to the type of a field, it means that if the server resolves that field to `null`, the response will *fail validation*.

Q5: How to do *Error Handling* in GraphQL? ☆☆

Topics: GraphQL

Answer:

A successful GraphQL query is supposed to return a JSON object with a root field called `"data"`. If the request fails or partially fails (e.g. because the user requesting the data doesn't have the right access permissions), a second root field called `"errors"` is added to the response:

```
{
  "data": { ... },
  "errors": [ ... ]
}
```

Q6: Where is GraphQL *useful*? ☆☆

Topics: GraphQL

Answer:

GraphQL helps where your **client needs a flexible response** format to avoid extra queries and/or massive data transformation with the overhead of keeping them in sync.

Using a GraphQL server makes it very easy for a client side developer to change the response format without any change on the backend.

With GraphQL, you can describe the required data in a more natural way. It can speed up development, because in application structures like **top-down rendering** in React, the required data is more similar to your component structure.

Q7: What is GraphQL `schema`? ☆☆

Topics: GraphQL

Answer:

Every GraphQL server has two core parts that determine how it works: a schema and resolve functions.

The *schema* is a model of the data that can be fetched through the GraphQL server. It defines what queries clients are allowed to make, what types of data can be fetched from the server, and what the relationships between these types are.

Consider:

```
type Author {
  id: Int
  name: String
  posts: [Post]
}
type Post {
```

```
  id: Int
  title: String
  text: String
  author: Author
}
type Query {
  getAuthor(id: Int): Author
  getPostsByTitle(titleContains: String): [Post]
}
schema {
  query: Query
}
```

Q8: What is difference between `Mutation` and `Query`? ☆☆

Topics: GraphQL

Answer:

Technically any GraphQL *query* could be implemented to cause a data write. But there is a convention that any operations that cause writes should be sent explicitly via a *mutation*.

Besides the difference in the semantic, there is one important technical difference:

Query fields can be executed in parallel by the GraphQL engine while Mutation top-level fields MUST execute serially according to the spec.

Q9: How to do *Server-Side Caching* in GraphQL? ☆☆☆

Topics: GraphQL

Answer:

One common concern with GraphQL, especially when comparing it to REST, are the difficulties to maintain server-side cache. With REST, it's easy to cache the data for each endpoint, since it's sure that the *structure* of the data will not change.

With GraphQL on the other hand, it's not clear what a client will request next, so putting a caching layer right behind the API doesn't make a lot of sense.

Server-side caching still is a challenge with GraphQL.

Q10: How to do `Authentication` and `Authorization` in GraphQL? ☆☆☆

Topics: GraphQL

Answer:

Authentication and authorization are often confused. *Authentication* describes the process of claiming an *identity*. That's what you do when you log in to a service with a username and password, you authenticate yourself. *Authorization* on the other hand describes *permission rules* that specify the access rights of individual users and user groups to certain parts of the system.

Authentication in GraphQL can be implemented with common patterns such as OAuth.

To implement authorization, it is recommended to delegate any data access logic to the business logic layer and not handle it directly in the GraphQL implementation.

Q11: Does GraphQL support *offline* usage? ☆☆☆

Topics: GraphQL

Answer:

GraphQL is a query language for (web) APIs, and in that sense by definition only works online. However, offline support on the client-side is a valid concern. The caching abilities of Relay and Apollo might already be enough for some use cases, but there isn't a popular solution for actually persisting stored data yet.

Q12: List the *key concepts* of the GraphQL query language ☆☆☆

Topics: GraphQL

Answer:

Key concepts of the GraphQL query language are:

- Hierarchical
- Product-centric
- Strong-typing
- Client-specified queries
- Introspective

Q13: Explain the main *difference* between REST and GraphQL ☆☆☆

Topics: GraphQL

Answer:

The main and most important difference between REST and GraphQL is that *GraphQL is not dealing with dedicated resources, instead everything is regarded as a graph and therefore is connected and can be queried to app exact needs*.

Q14: What kind of *operations* could GraphQL *schema* have? ☆☆☆

Topics: GraphQL

Answer:

There are three kinds of operation available at the entry points to the schema:

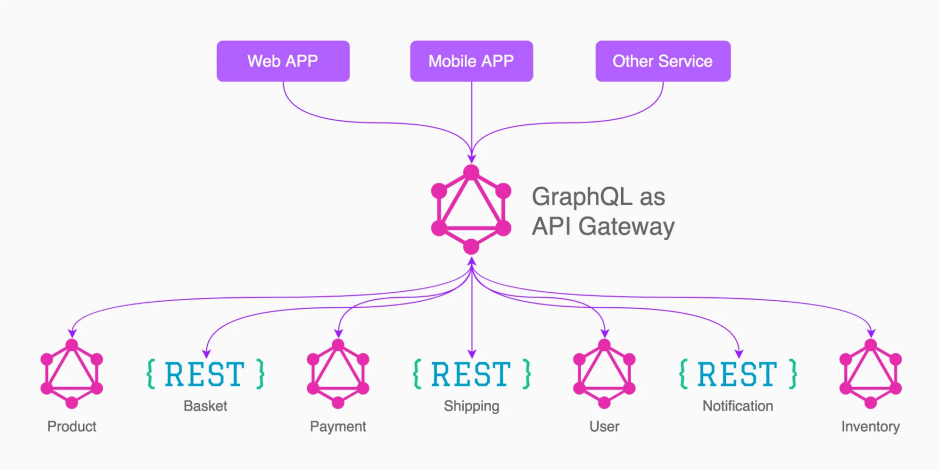
- Schema must have `query` object type whose fields are the set of query endpoints of the server application.
- A schema may optionally have `mutation` object type, whose fields are the set of mutations available on the server.
- The schema may have a `subscription` object type, whose fields are the set of subscriptions.

Q15: Whether do you find GraphQL the right fit for designing *microservice* architecture? ☆☆☆

Topics: Microservices GraphQL

Answer:

GraphQL and microservices are a perfect fit, because GraphQL hides the fact that you have a microservice architecture from the clients. From a backend perspective, you want to split everything into microservices, but from a frontend perspective, you would like all your data to come from a single API. Using GraphQL is the best way I know of that lets you do both. It lets you split up your backend into microservices, while still providing a single API to all your application, and allowing joins across data from different services.



Q16: Are there any *disadvantages* to GraphQL? ☆☆☆☆

Topics: GraphQL

Answer:

Disadvantages:

- You **need to learn** how to set up GraphQL. The ecosystem is still rapidly evolving so you have to keep up.
- You need to send the queries from the client, you can just send strings but if you want more comfort and caching you'll use a client library -> **extra code** in your client
- You need to define the schema beforehand => **extra work** before you get results
- You need to have a graphql endpoint on your server => **new libraries** that you don't know yet
- Graphql queries are **more bytes** than simply going to a REST endpoint
- The server needs to do **more processing** to parse the query and verify the parameters

Q17: Can you make a GraphQL type both an `input` and `output` type? ☆☆☆☆

Topics: GraphQL

Answer:

In the GraphQL spec, objects and input objects are distinct things. While an implementation might *provide convenience code* (see below) to create an object and a corresponding input object from a single definition, under the covers, the spec indicates that they'll have to be separate things.

Consider:

```
const RelativeTemplate = name => {
  return {
    name: name,
```

```
    fields: () => ({
      name: { type: GraphQLString },
      reference: { type: GraphQLString }
    })
  };
};

const RelativeType = {
  input: new GraphQLInputObjectType(RelativeTemplate("RelativeInput")),
  output: new GraphQLObjectType(RelativeTemplate("RelativeOutput"))
};
```

Q18: How to implement a *set* of GraphQL mutations in *single* transaction? ☆☆☆☆

Topics: GraphQL

Answer:

GraphQL has a mutation root object, which can have one or more top level fields. These top level fields are the individual mutations. If we want a service with transaction behavior, we'll implement it as a single mutation. The back-end implementation has to deal with how to ensure transactional behavior.

It's important to denote the difference between *mutations* and *requests*, especially regarding concurrency. Each request performs the individual mutations of that request in serial, but multiple requests can be processed concurrently.

Q19: How do you prevent *nested attack* on GraphQL server? ☆☆☆☆☆

Topics: GraphQL

Problem:

Consider the query:

```
{
  authors {
    firstName
    posts {
      title
      author {
        firstName
        posts{
          title
          author {
            firstName
            posts {
              title
              [n author]
                [n post]
            }
          }
        }
      }
    }
  }
}
```

In other words, how can you limit the number of recursions being submitted in a query?

Solution:

I'll just list all of the different methods:

- **Query validation** - in every GraphQL server, the first step to running a query is validation - this is where the server tries to determine if there are any serious errors in the query, so that we can avoid using actual server resources
- **Query timeout** - if it's not possible to detect that a query will be too resource-intensive statically (perhaps even shallow queries can be very expensive!), then we can simply add a timeout to the query execution.
- **Query whitelisting** - you could compile a list of allowed queries ahead of time, and check any incoming queries against that list
- **Query cost limiting** - similar to query timeouts, you can assign a cost to different operations during query execution, for example a database query, and limit the total cost the client is able to use per query

(1) and (2) in particular are probably something every GraphQL server should have by default, especially since many new developers might not be aware of these concerns.

Also there are some Node.js implementations that impose cost and depth bounds on incoming GraphQL documents.

- graphql-depth-limit
- graphql-validation-complexity
- graphql-query-complexity

Q20: Is it possible to use *inheritance* with GraphQL input types?

☆☆☆☆☆

Topics: GraphQL

Answer:

No, the spec does not allow input types to implement interfaces. And GraphQL type system in general does not define any form of inheritance (the extends keyword adds fields to an existing type, and isn't for inheritance).

Better yet, you might be able to solve your problem via composition (always keep *composition over inheritance* in mind).

```
input Name {
  firstName: String
  lastName: String
}

input UserInput {
  name: Name
  password: String!
}

input UserChangesInput {
  name: Name
  id: ID!
  password: String
}
```

FullStack.Cafe - Kill Your Tech Interview

Q1: How to query *all* the GraphQL type fields without writing a long query? ☆☆☆

Topics: GraphQL

Answer:

Unfortunately what you'd like to do is not possible. GraphQL requires you to be explicit about specifying which fields you would like returned from your query.

Q2: What is `AST` in GraphQL? ☆☆☆☆☆

Topics: GraphQL

Answer:

AST stands for **Abstract Syntax Tree**. When a GraphQL server receives a query to process it generally comes in as a String. This string must be tokenised and parsed into a representation that the machine understands. This representation is called *an abstract syntax tree*.

When GraphQL Processes the query, it walks the tree executing each part against the schema.

Q3: How to respond with *different status codes* in GraphQL? ☆☆☆☆☆

Topics: GraphQL

Answer:

The way to return errors in GraphQL (at least in graphql-js) is to throw errors inside the resolve functions. Because HTTP status codes are specific to the HTTP transport and GraphQL doesn't care about the transport, there's no way for you to set the status code there. What you can do instead is throw a specific error inside your resolve function:

```
age: (person, args) => {
  try {
    return fetchAge(person.id);
  } catch (e) {
    throw new Error("Could not connect to age service");
  }
}
```

GraphQL errors get sent to the client in the response like so:

```
{
  "data": {
    "name": "John",
    "age": null
  },
```

```
    "errors": [
      { "message": "Could not connect to age service" }
    ]
  }
```

And if the message is not enough information, you could create a special error class for your GraphQL server which includes a status code.

Q4: What the criteria set is for deciding when to use *GraphQL* vs. *HATEOAS*? ☆☆☆☆☆

Topics: GraphQL

Answer:

One interesting comparison is that people use GraphQL as a frontend for REST APIs, but no-one in their right mind would consider doing the converse. If you go for a federated/microservices design, so one GraphQL server fronts for others, they can use a common specification of the API between the frontend and the microservices; this is less certainly true if the microservices are REST.

The pros and cons of each are:

GraphQL

Pro:

- provides fine control of returned data in avoiding unneeded traffic
- eliminates needing to go back to the well over and over for attached / "follow-on" data
- following from the above, it allows the software designer to provide excellent performance by reducing latency - each query specifies all the things it needs, and the GraphQL implementation can assemble and deliver it with only one client<->server transaction
- possibility of slow deprecations instead of versioned APIs
- it's a query language
- introspection is built-in

Con:

- does not deal with caching (although there are now libraries that will take care of this)

HATEOAS / REST

Pro:

- caching is a well-understood matter
- relatively mature and well-understood
- lots of infrastructure for eg CDNs to spread the load
- very suitable for microservices
- file uploads are possible

Con:

- the "back to the well" problem
- not as rigidly specified
- each implementation of server and client(s) must make its own decisions
- querying is not standard

Q5: How would you model *recursive* data structures in GraphQL?

☆☆☆☆☆

Topics: GraphQL

Problem:

I have a tree data structure that I would like to return via a GraphQL API. I have modeled the structure as something like:

```
type Tag{
    id: String!
    children: [Tag]
}
```

The problem appears when one wants to get the tags to an arbitrary depth. To get all the children to (for example) level 3 one would write a query like:

```
{
    tags {
        id
        children {
            id
            children {
                id
            }
        }
    }
}
```

Is there a way to write a query to return all the tags to an arbitrary depth?

Solution:

The idea is to flatten the tree on data level using IDs to reference them (each child with it's parent) and marking the roots of the tree, then on client side build up the tree again recursively.

```
type Query {
    tags: [Tag]
}

type Tag {
    id: ID!
    children: [ID]
    root: Boolean
}
```

Response:

```
{
    "tags": [
        {"id": "1", "children": ["2"], "root": true},
        {"id": "2", "children": [], "root": false}
    ]
}
```