GraphQL

1. Write a short note on GraphQL.

=>

GraphQL is a query language for API.

It is a specification 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.

Companies like Facebook, GitHub, Coursera, Myntra, etc uses GraphQL.

It is a language that can be taught to a software client application.

The application can then communicate with a backend service, also speaking GraphQL, to request data.

The language is close to JSON and has operations for reading (queries) and writing (mutations) data.

The operations are strings for which the GraphQL service can then respond in desired format, often JSON.

It is a run time, a layer, written in any language for a server application to understand- and respond to GraphQL requests.

Key features of GraphQL are:

1. **Hierarchical:** The structure of the queries look exactly like the response data.
2. **Strong-typing:** The presence of a type system allows for syntactic query validation and the server responds with appropriate error messages.
3. **Client-specified queries:** The client decides what data they want through different fields in the queries.
4. **Introspective:** GraphQL allows the client to inquire about the fields, types, and supported queries. This is what allows GraphQL to perform query autocompletion and provide schema documentation.

2. Discuss on how GraphQL is the better REST.

=>

Over the past decade, REST has become the standard for designing web APIs.

It offers some great ideas, such as *stateless servers* and *structured access to resources*. However, REST APIs have shown to be too inflexible to keep up with the rapidly changing requirements of the clients that access them.

GraphQL was developed to cope with the need for more flexibility and efficiency!

It solves many of the shortcomings and inefficiencies that developers experience when interacting with REST APIs.

Some similarities between GraphQL and REST:

1. Both have the idea of a resource, and can specify IDs for those resources.
2. Both can be fetched via an HTTP GET request with a URL.
3. Both can return JSON data in the request.
4. The list of endpoints in a REST API is similar to the list of fields on the Query and Mutation types in a GraphQL API. They are both the entry points into the data.
5. Endpoints in REST and fields in GraphQL both end up calling functions on the server.

Some differences between GraphQL and REST:

1. In REST, the endpoint you call is the identity of that object. In GraphQL, the identity is separate from how you fetch it.
2. In REST, the shape and size of the resource is determined by the server. In GraphQL, the server declares what resources are available, and the client asks for what it needs at the time.
3. In GraphQL, you can traverse from the entry point to related data, following relationships defined in the schema, in a single request. In REST, you have to call multiple endpoints to fetch related resources.
4. In REST, each request usually calls exactly one route handler function. In GraphQL, one query can call many resolvers to construct a nested response with multiple resources.
5. In REST, you specify a write by changing the HTTP verb from GET to something else like POST. In GraphQL, you change a keyword in the query.

3. What is GraphQL SDL?

=>

A GraphQL Schema Definition is the most concise way to specify a GraphQL schema. The syntax is well-defined and are part of the official GraphQL specification.

SDL consists of - Field, Types, Arguments, Queries, Mutation.

Just like database schemas, GraphQL schemas define the structure of our data. This includes the types of data and the kind of operations that can be performed on the data. Consider the following schema that uses the GraphQL query language to define an author:

```
type Author {
id: ID!
Info: Person
}
```

```
type Person {
name: String!
age: Int
gender: String
}
```

Type:

Here, Author is a **type**.

A type has a name:

```
type Name {
  # ...
}
```

Fields:

We are defining author with following **fields**

id and info.

A field has a name and a type:

GraphQL comes with a set of default scalar types, they are as follows:

Float: A signed double precision floating point value.

Int: A signed 32-bit integer.

String: A sequence of characters.

Boolean: true or false

ID: Unique identifier that ma be either an integer or string.

Arguments

Every field on a GraphQL object type can have zero or more arguments.

{

```
 user(id: 1) {
   name
 }
}
```

Here, user is returned with the id as 1, where id is an argument.

The Query and Mutation types:

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

Every GraphQL service has a query type and may or may not have a mutation type, that define the entry point of every GraphQL query.

4. Explain Queries and Mutations in GraphQL.

=>

Queries:

Queries are used by the client to request the data it needs from the server. Unlike REST APIs where there's a clearly defined structure of information returned from each endpoint, GraphQL always exposes only one endpoint, allowing the client to decide what data it really needs from a predefined pattern.

Mutations:

Mutations is the key. In GraphQL, mutations are used to **CUD**:

- **C**reate new data
- **U**pdate existing data
- **D**elete existing data

The syntax for mutations looks almost the same as queries, but they must start with the mutation keyword.

Read Operation	
REST API Endpoint	http://localhost:9000/api/Employee
HTTP method	GET
REST API Response	[{ "EmployeeID": 1, "Name": "demo",

	"ManagerID": 1
	},
	{
	"EmployeeID": 2,
	"Name": "abcd",
	"ManagerID": 1
	}
	]	

Query	query{
	Employees{
	EmployeeID
	Name
	}
	}
		The GraphQL query to read data as needed. Here only EmployeeID and Name is fetched from REST API using GraphQL API.

Schema	type Query {
	Employees: [Employee]
	}
	
	type Employee {
	EmployeeID: ID
	Name: String
	ManagerID: Int
	}
		GraphQL schema is a contract between the server and client.
		
		It determines the shape of the queries and mutations that will be accepted by the server.

Resolver	Employees: () => {
	return fetch('http://localhost:9000/api/Employee')
	.then(response => response.json())
	}
		A function on a GraphQL server that's responsible for fetching the data for a single field.
		Here, the resolver fetches the data from the REST API and returns to the Employees field.

Result	{
	"data": {
	"Employees": [	

	{
	"EmployeeID": "1",
	"Name": "demo"
	},
	{
	"EmployeeID": "2",
	"Name": "abcd"
	}
	]
	}
	}	

Create Operation		
REST API Endpoint	http://localhost:9000/api/Employee	
HTTP method	POST	

Query	mutation {
	createEmployee(input: {EmployeeID: 1010, Name: "xyz"})
	}
		The GraphQL query to create data passed.

Schema	type Mutation {
	createEmployee(input : EmployeeInput!): String
	}
	
	input EmployeeInput {
	EmployeeID: ID
	Name: String
	}
		Mutations follow the same syntactical structure as queries, but they always need to start with the mutation keyword.

Resolver	createEmployee: (input) => {
	var answer = JSON.stringify(input);
	answer = JSON.parse(answer);
	var EmployeeID = answer.input.EmployeeID;
	var Name = answer.input.Name;
	
	return fetch(' http://localhost:9000/api/Employee', {
	method: 'POST',
	body: JSON.stringify({
	"EmployeeID": EmployeeID,
	"Name": Name
	}),
	headers: {
	"Content-type": "application/json"
	}
	})
		Here the resolver function adds a new Employee on REST API server using the GraphQL API.

	.then(response => (response.status == "204") ? "Record inserted" : "Record not inserted")	

Result	{
	"data": {
	"createEmployee": "Record inserted"
	}
	}	

Update Operation		
REST API Endpoint	http://localhost:9000/api/Employee	
HTTP method	PUT	

Query	mutation {
	updateEmployee(input: {EmployeeID: 1010, Name: "xyz"})
	}
		The GraphQL query to update data passed.

Schema	type Mutation {
	updateEmployee(input : EmployeeInput!): String
	}
	
	input EmployeeInput {
	EmployeeID: ID
	Name: String
	}
		Mutations follow the same syntactical structure as queries, but they always need to start with the mutation keyword.

Resolver	updateEmployee: (input) => {
	var answer = JSON.stringify(input);
	answer = JSON.parse(answer);
	var EmployeeID = answer.input.EmployeeID;
	var Name = answer.input.Name;
	
	return fetch(' http://localhost:9000/api/Employee', {
	method: 'PUT',
	body: JSON.stringify({
	"EmployeeID": EmployeeID,
	"Name": Name
	}),
	headers: {
	"Content-type": "application/json"
	}
	})
	.then(response => (response.status == "204") ? "Record updated" : "Record not updated")
		Here the resolver function edits an Employee on REST API server using the GraphQL API.

Result	{	

	"data": {
	"updateEmployee": "Record updated"
	}
	}	

Delete Operation		
REST API Endpoint	http://localhost:9000/api/Employee	
HTTP method	DELETE	

Query	mutation {
	deleteEmployee(input: {EmployeeID: 1010, Name: "xyz"})
	}
		The GraphQL query to delete data passed.

Schema	type Mutation {
	deleteEmployee(input : EmployeeInput!): String
	}
	
	input EmployeeInput {
	EmployeeID: ID
	Name: String
	}
		Mutations follow the same syntactical structure as queries, but they always need to start with the mutation keyword.

Resolver	deleteEmployee: (input) => {
	var answer = JSON.stringify(input);
	answer = JSON.parse(answer);
	var EmployeeID = answer.input.EmployeeID;
	var Name = answer.input.Name;
	
	return fetch(' http://localhost:9000/api/Employee', {
	method: 'PUT',
	body: JSON.stringify({
	"EmployeeID": EmployeeID,
	"Name": Name
	}),
	headers: {
	"Content-type": "application/json"
	}
	})
	.then(response => (response.status == "204") ? "Record deleted" : "Record not deleted")
		Here the resolver function edits an Employee on REST API server using the GraphQL API.

Result	{
	"data": {
	"deleteEmployee": "Record deleted"
	}
	}	

5. Write a note on Authentication and Authorization in GraphQL?

=>

Authentication is the process of determining claimed user identity by checking user-provided credentials.

For instance, to log in to an application, a user must provide some kind of credential with which to identify themselves.

These credentials can be a username/password pair or a kind of token.

If the credentials match what's in the application database, the user is logged in.

Authentication is the process of recognizing a user's identity. It is different from authorization.

Authentication and authorization are commonly used together as they work hand-in-hand but they are completely different.

Authorization occurs after a successful authentication, it checks the access levels or privileges of the user which will determine what the user can see or do with the application.

Authorization is a type of business logic that describes whether a given user/session/context has permission to perform an action or see a piece of data.

Enforcing this kind of behaviour should happen in the business logic layer. It is tempting to place authorization logic in the GraphQL layer like so:

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 or by using JSON Web Tokens.

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.

6. Enlist the programming languages and libraries to work with GraphQL server-side.

=>

The following are the programming languages and libraries to work with GraphQL server-side:

1. C# / .NET

i) graphql-dotnet: GraphQL for .NET

ii) graphql-net: Convert GraphQL to IQueryable

2. Java

i) graphql-java: A Java library for building GraphQL APIs.

3. JavaScript

i) GraphQL.js: The reference implementation of the GraphQL specification, designed for running GraphQL in a Node.js environment.

ii) express-graphql: The reference implementation of a GraphQL API server over an Express webserver.

iii) apollo-server: A set of GraphQL server packages from Apollo that work with various Node.js HTTP frameworks (Express, Connect, Hapi, Koa etc).

4. PHP

i) graphql-php: A PHP port of GraphQL reference implementation

ii) graphql-relay-php: A library to help construct a graphql-php server supporting react-relay.

5. Python

i) Graphene: A Python library for building GraphQL APIs.

7. Write a note on "GraphiQL".

=> GraphiQL is an in-browser IDE for exploring GraphQL.

If you using a node.js server, just use express-graphql and it can automatically present GraphiQL.

If you are using another GraphQL service, then GraphiQL is pretty easy to set up.

With npm: npm install --save graphiql

Features:

- Syntax highlighting
- Intelligent type ahead of fields, arguments, types, and more.
- Real-time error highlighting and reporting.
- Automatic query completion.
- Run and inspect query results.

It displays a code editor with autocomplete on the far left, comes with error highlighting, query results in the middle column, and a documentation explorer on the far right.

The Query Variables on the bottom left show what is actually being manipulated in the request.

For a developer consumer, a GraphiQL interface could be helpful for building GraphQL queries, viewing the types of data that are available, and more.

8. Write a note on Prisma.

=>

Prisma is a GraphQL database proxy. It turns your database into a GraphQL API which can be consumed by your resolvers via GraphQL bindings.

Prisma is a standalone component which is deployed in front of your SQL database and generates a GraphQL API.

To get started and create a new GraphQL API with Prisma, you simply define your data model using GraphQL SDL and use the Prisma CLI to deploy your changes.

It has the two important benefits:

1. Read & write to your database with GraphQL.
2. Easy data modeling and migrations with GraphQL

To get started with Prisma follow the below steps:

1. Install Prisma:
   npm install -g prisma

2. Connect your database or create a new database with Prisma CLI.

3. Define your data model:
   Use GraphQL SDL to define the structure of the database.

4. Deploy the Prisma GraphQL API:
   Once the data model is ready, deploy your changes.
   >_ prisma deploy
   Prisma then generates a powerful GraphQL API for your database.

5. Use the API to build a GraphQL server:
   You can now use GraphQL to read and write data.

9. How REST and GraphQL are different? Explain.

=>

Same as Answer to the Q2. i.e. **Discuss on how GraphQL is the better REST.**

10. Using schema definition language (SDI) write schema definition for the following JSON, where the types in the JSON are Students and Marks

```
allStudents:[
{
   "name":"Raj",
   "marks":
   [
   {
      "english":
      "maths":
      "science":
      "hindi":
      "ss":
   }
   ]
},
{
   "name":"Raj",
   "marks":
   [
   {
      "english":
      "maths":
      "science":
      "hindi":
      "ss":
   }
   ]
},
```

```
]
=>
    type Students {
        name: String!
        marks: [Marks]
    }

    type Marks {
        english: String
        maths: String
        science: String
        hindi: String
        ss: String
    }
```

All the Best!