Core Concepts

In this chapter, you'll learn about some fundamental language constructs of GraphQL. That includes a first glimpse at the syntax for defining *types* as well as sending *queries* and *mutations*.

The Schema Definition Language (SDL)

GraphQL has its own type system that's used to define the *schema* of an API. The syntax for writing schemas is called Schema Definition Language (SDL).

Here is an example of how we can use the SDL to define a simple type called `Person`:

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

This type has two *fields*, they're called `name` and `age` and are respectively of type `String` and `Int`. The `!` following the type means that this field is *required*.

It's also possible to express relationships between types. In the example of a *blogging* application, a `Person` could be associated with a `Post`:

```
type Post {
  title: String!
  author: Person!
}
```

Conversely, the other end of the relationship needs to be placed on the `Person` type:

```
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
```

Note that we just created a *one-to-many*-relationship between `Person` and `Post` since the `posts` field on `Person` is actually an *array* of posts.

Fetching Data with Queries

When working with REST APIs, data is loaded from specific endpoints. Each endpoint has a clearly defined structure of the information that it returns. This means that the data requirements of a client are effectively *encoded* in the URL that it connects to.

The approach that's taken in GraphQL is radically different. Instead of having multiple endpoints that return fixed data structures, GraphQL APIs typically only expose *a single endpoint*. This works because the structure of the data that's returned is not fixed. Instead, it's completely flexible and lets the client decide what data is actually needed.

That means that the client needs to send more *information* to the server to express its data needs - this information is called a *query*.

Note: Unfortunately, we no longer provide the *Run in Sandbox* feature that is demonstrated in the video at 13:50. We are really sorry for the inconvenience.

Basic Queries

Let's take a look at an example query that a client could send to a server:

```
{
  allPersons {
    name
  }
}
```

The `allPersons` field in this query is called the *root field* of the query. Everything that follows the root field, is called the *payload* of the query. The only field that's specified in this query's payload is `name`.

This query would return a list of all persons currently stored in the database. Here's an example response:

```
{
  "allPersons": [
    { "name": "Johnny" },
    { "name": "Sarah" },
    { "name": "Alice" }
```

```
  ]
}
```

Notice that each person only has the `name` in the response, but the `age` is not returned by the server. That's exactly because `name` was the only field that was specified in the query.

If the client also needed the persons' `age`, all it has to do is slightly adjust the query and include the new field in the query's payload:

```
{
  allPersons {
    name
    age
  }
}
```

One of the major advantages of GraphQL is that it allows for naturally querying *nested* information. For example, if you wanted to load all the `posts` that a `Person` has written, you could simply follow the structure of your types to request this information:

```
{
  allPersons {
    name
    age
    posts {
      title
    }
  }
}
```

Queries with Arguments

In GraphQL, each *field* can have zero or more arguments if that's specified in the *schema*. For example, the `allPersons` field could have a `last` parameter to only return up to a specific number of persons. Here's what a corresponding query would look like:

```
{
  allPersons(last: 2) {
    name
```

```
  }
}
```

Writing Data with Mutations

Next to requesting information from a server, the majority of applications also need some way of making changes to the data that's currently stored in the backend. With GraphQL, these changes are made using so-called *mutations*. There generally are three kinds of mutations:

- creating new data
- updating existing data
- deleting existing data

Mutations follow the same syntactical structure as queries, but they always need to start with the `mutation` keyword. Here's an example for how we might create a new `Person`:

```
mutation {
  createPerson(name: "Bob", age: 36) {
    name
    age
  }
}
```

Notice that similar to the query we wrote before, the mutation also has a *root field* - in this case it's called `createPerson`. We also already learned about the concepts of arguments for fields. In this case, the `createPerson` field takes two arguments that specify the new person's `name` and `age`.

Like with a query, we're also able to specify a payload for a mutation in which we can ask for different properties of the new `Person` object. In our case, we're asking for the `name` and the `age` - though admittedly that's not super helpful in our example since we obviously already know them as we pass them into the mutation. However, being able to also query information when sending mutations can be a very powerful tool that allows you to retrieve new information from the server in a single roundtrip!

The server response for the above mutation would look as follows:

```
"createPerson": {
  "name": "Bob",
```

```
  "age": 36,
}
```

One pattern you'll often find is that GraphQL types have unique *IDs* that are generated by the server when new objects are created. Extending our `Person` type from before, we could add an `id` like this:

```
type Person {
  id: ID!
  name: String!
  age: Int!
}
```

Now, when a new `Person` is created, you could directly ask for the `id` in the payload of the mutation, since that is information that wasn't available on the client beforehand:

```
mutation {
  createPerson(name: "Alice", age: 36) {
    id
  }
}
```

Realtime Updates with Subscriptions

Another important requirement for many applications today is to have a *realtime* connection to the server in order to get immediately informed about important events. For this use case, GraphQL offers the concept of *subscriptions*.

When a client *subscribes* to an event, it will initiate and hold a steady connection to the server. Whenever that particular event then actually happens, the server pushes the corresponding data to the client. Unlike queries and mutations that follow a typical "*request-response*-cycle", subscriptions represent a *stream* of data sent over to the client.

Subscriptions are written using the same syntax as queries and mutations. Here's an example where we subscribe on events happening on the `Person` type:

```
subscription {
  newPerson {
    name
    age
```

```
  }
}
```

After a client sent this subscription to a server, a connection is opened between them. Then, whenever a new mutation is performed that creates a new `Person`, the server sends the information about this person over to the client:

```
{
  "newPerson": {
    "name": "Jane",
    "age": 23
  }
}
```

Defining a Schema

Now that you have a basic understanding of what queries, mutations, and subscriptions look like, let's put it all together and learn how you can write a schema that would allow you to execute the examples you've seen so far.

The *schema* is one of the most important concepts when working with a GraphQL API. It specifies the capabilities of the API and defines how clients can request the data. It is often seen as a *contract* between the server and client.

Generally, a schema is simply a collection of GraphQL types. However, when writing the schema for an API, there are some special *root* types:

```
type Query { ... }
type Mutation { ... }
type Subscription { ... }
```

The `Query`, `Mutation`, and `Subscription` types are the *entry points* for the requests sent by the client. To enable the `allPersons`-query that we saw before, the `Query` type would have to be written as follows:

```
type Query {
  allPersons: [Person!]!
}
```

`allPersons` is called a *root field* of the API. Considering again the example where we added the `last` argument to the `allPersons` field, we'd have to write the `Query` as follows:

```
type Query {
  allPersons(last: Int): [Person!]!
}
```

Similarly, for the `createPerson` -mutation, we'll have to add a root field to the `Mutation` type:

```
type Mutation {
  createPerson(name: String!, age: Int!): Person!
}
```

Notice that this root field takes two arguments as well, the `name` and the `age` of the new `Person` .

Finally, for the subscriptions, we'd have to add the `newPerson` root field:

```
type Subscription {
  newPerson: Person!
}
```

Putting it all together, this is the *full* schema for all the queries and mutation that you have seen in this chapter:

```
type Query {
  allPersons(last: Int): [Person!]!
  allPosts(last: Int): [Post!]!
}

type Mutation {
  createPerson(name: String!, age: Int!): Person!
  updatePerson(id: ID!, name: String!, age: Int!): Person!
  deletePerson(id: ID!): Person!
}

type Subscription {
  newPerson: Person!
}

type Person {
  id: ID!
  name: String!
```

```
  age: Int!
  posts: [Post!]!
}

type Post {
  title: String!
  author: Person!
}
```