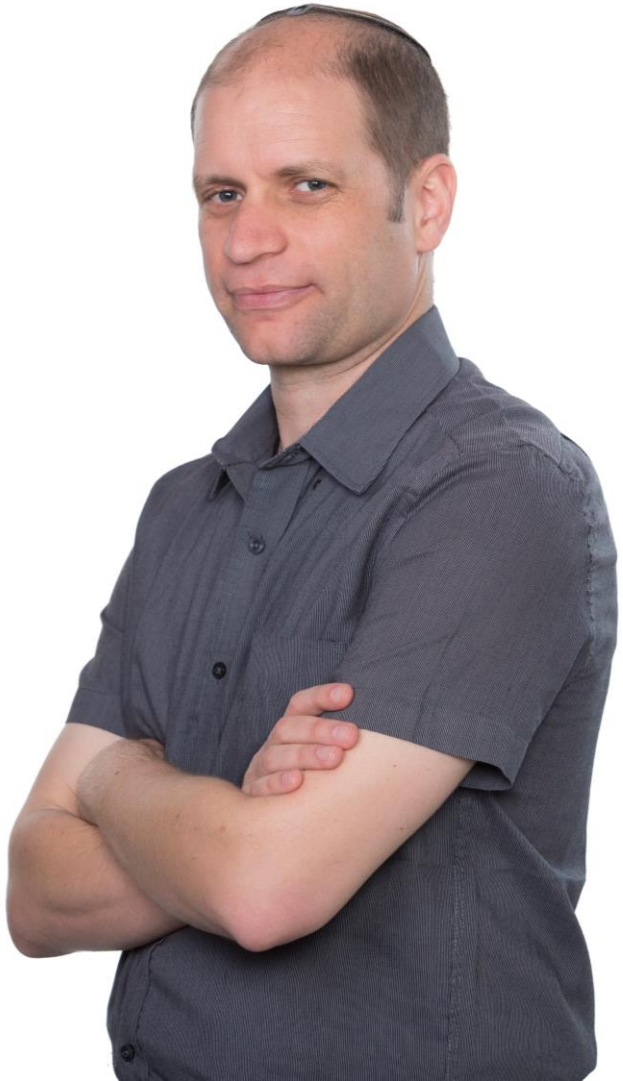
GraphQL Schema



Memi Lavi
www.memilavi.com

GraphQL Schema

- The first step when working with GraphQL is to define the schema
- GraphQL cannot work without schema
- We'll learn all about it in this section

Role of Schema

- As we saw, with GraphQL we can query specific fields from specific objects:

```
{
  books  {
    name,
    pages
  }
}
```

Role of Schema

- For that, GraphQL needs to know there is such an object

  and such fields:

```
{
  books {
    name,
    pages
  }
}
```

The book object

Name and pages fields

Role of Schema

- The schema is used to:

Define the shape of data

Define queries

Define mutations

Define subscriptions

Role of Schema

- GraphQL uses the schema to:

Validate queries

...and other operations

Provide auto complete in playground

Schema Type System

- The schema uses type system to define:

Field name

Field type

Nullability

Schema Language

- The schema uses specialized language

Schema Definition Language

- Defines the objects, fields, type system, queries and more

Schema Generation

- The schema is sometimes auto-generated by the GraphQL implementation
- Based on the concrete language-specific objects
- Can be viewed in the playground

Object Type and Fields

- The basic building blocks of the schema
- Describe the entities (=>Objects) and their properties (=>Fields) in the system
- Allow GraphQL to be familiar with the system's object and to provide validation to these entities

Object Type and Fields

- Let's look again at the schema of our app:

Object Type

```
type Book {
  bookId: Int!
  name: String!
  pages: Int!
}
```

Fields

Field Types

- Each field has a data type, indicating the type of data it can store
- GraphQL specifications contain built-in Scalar types
- Primitive types that hold a single value
- Complex types can be also defined

Scalar Types

- Built in scalar types:

Data Type	Description
Int	A signed 32-bit integer (64, 5890, -17 etc)

Scalar Types

- Built in scalar types:

Data Type	Description
Int	A signed 32-bit integer (64, 5890, -17 etc)
Float	A signed double precision floating-point value (18.99 etc)

Scalar Types

- Built in scalar types:

Data Type	Description
Int	A signed 32-bit integer (64, 5890, -17 etc)
Float	A signed double precision floating-point value (18.99 etc)
String	A string of characters UTF-8 encoded (“Hello world” etc)

Scalar Types

- Built in scalar types:

Data Type	Description
Int	A signed 32-bit integer (64, 5890, -17 etc)
Float	A signed double precision floating-point value (18.99 etc)
String	A string of characters UTF-8 encoded (“Hello world” etc)
Boolean	`true` or `false`

Scalar Types

- Built in scalar types:

Data Type	Description
Int	A signed 32-bit integer (64, 5890, -17 etc)
Float	A signed double precision floating-point value (18.99 etc)
String	A string of characters UTF-8 encoded (“Hello world” etc)
Boolean	`true` or `false`
ID	A unique identifier, identical to String

Field Types

- Most implementations add custom scalar types
- For example: `date`
- Let's see objects and fields in action

Nullable Fields

- As you probably noted, there's an exclamation mark near the type of fields in the schema

```
type Book {
  bookId: Int!
  name: String!
  pages: Int!
  price: Float!
  publishDate: DateTime!
}
```

Nullable Fields

- This indicates that the field is non-nullable – it can't be null
- If you'll try to create a new Book object using GraphQL and don't put values in all the fields – you'll get an error
- Can be changed using the object definition in the code

Enumerations

- A special type of scalar
- Limits the values a field can have to a predefined list
- Great for making sure only valid values are selected
- Invalid values will be marked as error in GraphQL

Lists

- Objects can hold lists of other objects
- Can be declared in the schema

Lists and Nulls

- You need to pay special attention to list and nullable types
- In our example, this is how it looks like:

```
reviews: [BookReview!]!
```

Note the two exclamation marks

Lists and Nulls

```
reviews: [BookReview!]!
```

The list cannot contain `null` items

There has to be a list (even an empty one). The `reviews` field cannot be null

- Let's see it in action

Unions

- Similar to interfaces
- Allow querying multiple object types in a single query
- In contrast to interface, object types do not have to have common fields
- Query can specify which field to retrieve from each object type