# GraphQL - Application Components

This chapter discusses different GraphQL components and the way they communicate with each other. The entire application components can be distinguished as below −

* Server-side Components
* Client-side Components

## Server-Side Components

GraphQL server forms the core component on the server side and allows to parse the queries coming from GraphQL client applications. Apollo Server is most commonly used implementation of GraphQL specification. Other server programming components include the following −

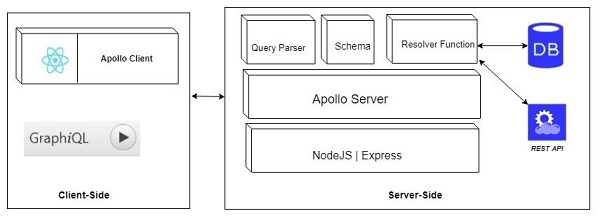
|  |  |
| --- | --- |
| **Sr.No.** | **Server Essentials & Description** |
| 1 | **Schema**  A GraphQL schema is at the center of any GraphQL server implementation and describes the functionality available to the clients which connect to it. |
| 2 | **Query**  A GraphQL query is the client application request to retrieve data from database or legacy API's. |
| 3 | **Resolver**  Resolvers provide the instructions for turning a GraphQL operation into data. They resolve the query to data by defining resolver functions. |

## Client-side Components

Given below are the client-side components −

|  |  |
| --- | --- |
| **Sr.No.** | **Tool & Description** |
| 1 | **GraphiQL**  Browser based interface for editing and testing GraphQL queries and mutations. |
| 2 | **ApolloClient**  Best tool to build GraphQL client applications. Integrates well with all javascript front-end. |

The below diagram shows a **Client-Server architecture**. The web server is built on NodeJs and Express framework. A request is made to the Apollo GraphQL Server by ReactJS application (built using Apollo Client library) or GraphiQL browser application. The query will be parsed and validated against a schema defined in the server. If the request schema passes the validation, then the associated resolver functions will be executed. The resolver will contain code to fetch data from an API or a database.



GraphQL Schema

The GraphQL schema is the core building block of GraphQL server implementation. The GraphQL server uses a GraphQL schema to define and describe the shape of your data graph. The GraphQL schema establishes the hierarchy of types with fields and also describes functionality available to the client applications. In simple words, you can say that "A schema is used to define a collection of types and the relationships between these types." It is also used to specify the exact queries and mutations which are available for clients to execute against your data graph.

[GraphQL](https://www.javatpoint.com/graphql) schema can be created by using any programming language, and we can also build an interface around it.

Here, we shall describe how to create a GraphQL schema for our GraphQL server and also the fundamental building blocks of the schema.

What shall we use?

* An Apollo server to execute GraphQL queries
* A makeExecutableSchema function in graphql-tools to bind schema and resolvers

makeExecutableSchema Function Syntax

The makeExecutableSchema function is used to take a single argument {} of Object type. Following is the syntax of makeExecutableSchema Function:

1. **import** { makeExecutableSchema } from 'graphql-tools';
2. **const** jsSchema = makeExecutableSchema({
3. typeDefs,
4. resolvers, // optional
5. logger, // optional
6. allowUndefinedInResolve = **false**, // optional
7. resolverValidationOptions = {}, // optional
8. directiveResolvers = **null**, // optional
9. schemaDirectives = **null**,  // optional
10. parseOptions = {},  // optional
11. inheritResolversFromInterfaces = **false**  // optional
12. });

Parameter Explanation

* **typeDefs:** This argument is used to represent a GraphQL query as a UTF-8 string. (Required argument)
* **Resolvers:** This argument has functions that are used to handle the query. (Optional argument)
* **logger:** This argument is used to print errors to the server console. (Optional argument)
* **parseOptions:** This argument is used to allow customization of parse when specifying typeDefs as a string. (Optional argument)
* **allowUndefinedInResolve:** This argument is set true by default. When you set it to false, it makes your resolve functions to throw errors if they return undefined.
* **resolverValidationOptions:** This argument is used to accept an object with Boolean properties. (Optional argument)
* **inheritResolversFromInterfaces:** This argument is used to accept a Boolean argument to check resolvers object inheritance. (Optional argument)

Supported types of GraphQL Schema

A type definition in a GraphQL schema must belong to one of the following categories:

* Scalar types
* Object types
* The Query type
* The Mutation type
* Input types

The performance and usage of the every declaration of the above fields is individually monitored by Apollo Studio. It also provides you the data that informs about the changes to your graph.

The Schema Definition Language

In GraphQL specification, there is a human-readable schema definition language (SDL) that is used to define your schema and store it as a string.

**For example:**

Let's see a simple schema example that defines two object types: Company and Owner:

1. type Company {
2. title: String
3. owner: Owner
4. }
5. type Owner {
6. name: String
7. companies: [Company]
8. }

In the above schema example, every Company has an owner, and every owner has a list of companies. When we define these relationships in a unified schema, we make client developers able to see the available data is and also request a specific subset of that data by using a single optimized query.

Note: The schema is not responsible for defining where data comes from or how data is stored. It is only responsible for the hierarchy implementation of types.

Example

Let's create a simple example to see how the schema works. In this example, we shall create a schema to query a list of employees from the server. The employee data will be stored in a flat-file, and we shall read data from the flat file by using a node module called **notarealdb**.

Download and Install Required Dependencies

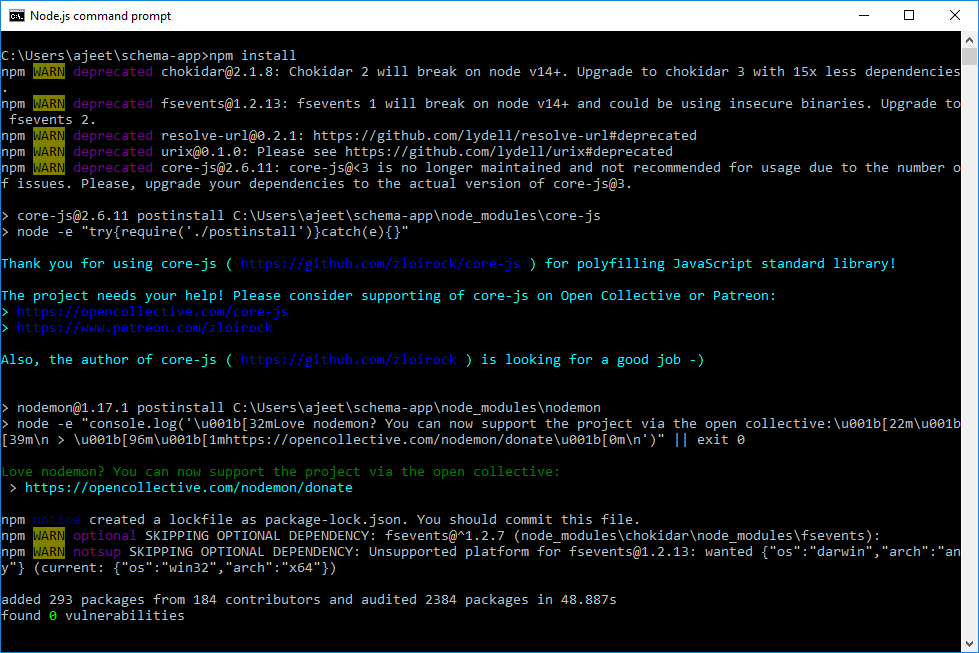
First, create a folder named "schema-app" and navigate to the folder by using node.js command prompt.

Then, Create a **package.json** file and install the dependencies:

1. {
2. "name": "hello-world-server",
3. "private": **true**,
4. "scripts": {
5. "start": "nodemon --ignore data/ server.js"
6. },
7. "dependencies": {
8. "apollo-server-express": "^1.4.0",
9. "body-parser": "^1.18.3",
10. "cors": "^2.8.4",
11. "express": "^4.16.3",
12. "graphql": "^0.13.2",
13. "graphql-tools": "^3.1.1"
14. },
15. "devDependencies": {
16. "nodemon": "1.17.1"
17. }
18. }

Now, install all dependencies by using the following command:

1. npm install



Create a Flat File Database in Data Folder

Here, we shall create flat files to store and retrieve data. First, create a folder named **data** and add two files **employees.json** and **companies.json**.

**companies.json file:**

1. [
2. {
3. "id": "com-101",
4. "name": "HCL",
5. "location": "Noida",
6. "rating":5.0
7. },
8. {
9. "id": "com-102",
10. "name": "TCS",
11. "location": "Mumbai",
12. "rating":4.5
13. }
14. ]

**employees.json file:**

1. [
2. {
3. "id": "E1001",
4. "firstName":"Ajeet",
5. "lastName":"Kumar",
6. "password": "pass123",
7. "companyId": "com-102"
8. },
9. {
10. "id": "E1002",
11. "firstName":"Mohan",
12. "lastName":"Bhargav",
13. "password": "pass123",
14. "companyId": "com-101"
15. },
16. {
17. "id": "E1003",
18. "firstName":"Rashmi",
19. "lastName":"Bansal",
20. "password": "pass123",
21. "companyId": "com-101"
22. }
23. ]

Create a Data Access Layer

Let's create a datastore to store the data folder content. Here, we use collection variables, employees, and companies.

Create a file **db.js** within the "schema-app" folder:

1. **const** { DataStore } = require('notarealdb');
2. **const** store = **new** DataStore('./data');
3. module.exports = {
4. employees:store.collection('employees'),
5. companies:store.collection('companies')
6. };

Create a Schema

Now, create a **schema.graphql** file in the "schema-app" folder and use the following code:

1. type Query {
2. hello:String
3. employees:[Employee]
4. }
5. type Employee {
6. id:ID!
7. firstName:String
8. lastName:String
9. password:String
10. companyId:String
11. }

Create Resolver

Create a file **resolvers.js** within the "schema-app" folder and use the following code:

1. **const** db = require('./db')
2. **const** Query = {
3. hello:() => {
4. **return** "Welcome to JavaTpoint..."
5. },
6. employees:() => db.employees.list()
7. }
8. module.exports = {Query}

Create Server.js and Configure GraphQL

1. **const** bodyParser = require('body-parser');
2. **const** cors = require('cors');
3. **const** express = require('express');
4. **const** db = require('./db');
5. **const** port = process.env.PORT || 4000;
6. **const** app = express();
7. **const** fs = require('fs')
8. **const** typeDefs = fs.readFileSync('./schema.graphql',{encoding:'utf-8'})
9. **const** resolvers = require('./resolvers')
10. **const** {makeExecutableSchema} = require('graphql-tools')
11. **const** schema = makeExecutableSchema({typeDefs, resolvers})
12. app.use(cors(), bodyParser.json());
13. **const**  {graphiqlExpress,graphqlExpress} = require('apollo-server-express')
14. app.use('/graphql',graphqlExpress({schema}))
15. app.use('/graphiql',graphiqlExpress({endpointURL:'/graphql'}))
16. app.listen(
17. port, () => console.info(
18. `Server started on port ${port}`
19. )
20. );

Run the Server

Run the server and use the following query in the editor:

1. {
2. hello
3. employees {
4. id
5. firstName
6. lastName
7. }
8. }

**Output:**

{

"data": {

"hello": "Welcome to JavaTpoint...."

"employees": [

{

"id": "E1001",

"firstName":"Ajeet",

"lastName":"Kumar",

},

{

"id": "E1002",

"firstName":"Mohan",

"lastName":"Bhargav",

},

{

"id": "E1003",

"firstName":"Rashmi",

"lastName":"Bansal",

}

]

}

}