Сообщение и мастер-класс по GraphQL

GraphQL это синтаксис, который описывает как запрашивать данные, и, в основном, используется клиентом для загрузки данных с сервера. GraphQL имеет три основные характеристики:

- Позволяет клиенту точно указать, какие данные ему нужны.
- Облегчает агрегацию данных из нескольких источников.
- Использует систему типов для описания данных.

GraphQL — это язык запросов с открытым исходным кодом, создавался как более эффективная альтернатива REST для разработки и использования программных интерфейсов приложений.
GraphQL обладает множеством достоинств, например:

1. Вы получаете информацию именно в том объёме, в котором запрашиваете..
2. Вам будет необходима всего одна конечная точка.
3. GraphQL — сильно типизированный язык, что позволяет предварительно оценить корректность запроса в рамках системы типов синтаксиса.

С чего начать?

Чтобы понять, как применять стандарт на практике, используем сервер в базовой конфигурации — Graphpack.

Нужно создать новую папку для проекта. В данном случае имя папки будет `graphql-server`, однако название не принципиально. Откройте терминал и введите:

```
mkdir graphql-server
```

На компьютере должен быть установлен npm или yarn. Перейдите в созданную папку и введите команду в зависимости от используемого менеджера:

```
npm init -y
```

или

```
yarn init
```

npm создаст файл `package.json`, в котором будут храниться все созданные вами зависимости и команды. Теперь нужно установить одну зависимость, которую мы будем использовать в рамках этой статьи.

Graphpack позволяет создать сервер GraphQL с базовой конфигурацией. Используя терминал, в корневой папке проекта установите Graphpack с помощью команды:

```
npm install --save-dev graphpack
```

Если вы используете yarn:

```
yarn add --dev graphpack
```

Перейдите к файлу package.json и добавьте следующий код:

```
"scripts": {

  "dev": "graphpack",

  "build": "graphpack build"

}
```

Создайте на сервере папку src. В этом примере это будет единственная папка на сервере, в которой необходимо создать три файла. В src создайте файл schema.graphql. В файл добавьте код:

```
type Query {

 hello: String

}
```

В этом файле будет находиться вся схема GraphQL.

Создайте второй файл в той же папке, назовите его resolvers.js. Разместите там следующий код:

```
import { users } from "./db";

const resolvers = {

 Query: {

  hello: () => "Hello World!"
```

```
  }

};

export default resolvers;
```

В этом файле будут размещены инструкции по выполнению операций GraphQL.

Создайте третий файл, db.js, содержащий код:

```
export let users = [

  { id: 1, name: "John Doe", email: "john@gmail.com", age: 22 },

  { id: 2, name: "Jane Doe", email: "jane@gmail.com", age: 23 }

];
```

Для тестирования работы GraphQL начинающим специалистам нет нужды использовать настоящие данные. Этот файл нужен для симуляции обращений к базе данных.

После выполнения операций папка src должна выглядеть следующим образом:

```
src

  |--db.js

  |--resolvers.js

  |--schema.graphql
```

Теперь выполните команду npm run dev для npm или yarn devдля yarn. Терминал должен вывести информацию об успешном запуске сервера.

Теперь можно перейти к localhost:4000. Система готова к работе над GraphQL API.

Схема

GraphQL прост для начинающих благодаря собственному языку — Schema Definition Language (SDL). SDL обладает интуитивно понятным синтаксисом и универсален для любой используемой технологии.

Типы

Типы — одна из основных особенностей языка запросов GraphQL. Это кастомные объекты, которые определяют, как будет выглядеть GraphQL API. Например, при разработке программного интерфейса для приложения, взаимодействующего с соцсетями, в API стоит объявить типы Posts, Users, Likes, Groups.

В типах есть поля, возвращающие определённые разновидности данных. Например, при создании типа User, в него стоит включить поля name, email, и age. Поля типов могут быть любыми и всегда возвращают данные в формате Int, Float, String, Boolean, ID, List of Object Types, или Custom Objects Types.

Чтобы создать первый тип, откройте файл schema.graphql и замените ранее прописанный там тип Query следующим кодом:

```
type User {

  id: ID!

  name: String!

  email: String!

  age: Int

}
```

Каждая запись типа User должна иметь идентификационный номер, поэтому поле id содержит данные соответствующего типа. Поля name и email содержат String (переменную типа строки символов), а age — целочисленную переменную.

Восклицательный знак в конце определения поля означает, что это поле не может быть пустым. Единственное поле без восклицательного знака — age.

Язык запросов GraphQL оперирует тремя основными концепциями:

1. *queries*, запросы — с их помощью получают данные с сервера.
2. *mutations*, изменения — модификация данных на сервере и их обновление.
3. *subscriptions*, подписки — методы поддержания постоянной связи с сервером.

Запросы

Откройте файл schema.graphql и добавьте тип Query:

```
type Query {

  users: [User!]!

}
```

Запрос users будет возвращать массив из одной и более записей типа User. Поскольку в определении использованы восклицательные знаки, ответ на запрос не может быть пустым.

Для получения конкретной записи User необходимо создать соответствующий запрос. В данном случает это будет запрос user в типе Query. Добавьте в код следующую строку:

```
user(id: ID!): User!
```

Теперь код должен выглядеть так:

```
type Query {

 users: [User!]!

 user(id: ID!): User!

}
```

Как видите, в запросах GraphQL можно передавать аргументы. В данном случае для получения конкретной записи в запросе в качестве аргумента используется её поле id.

Местонахождение данных, которые будут обрабатываться в соответствии с запросом, определяется в файле resolvers.js. Откройте этот файл и импортируйте учебную базу данных db.js:

```
import { users } from "./db";

const resolvers = {

  Query: {

    hello: () => "Hello World!"

  }
```

```
};

export default resolvers;
```

Затем замените функцию hello на user и users:

```
import { users } from "./db";

const resolvers = {

  Query: {

    user: (parent, { id }, context, info) => {

      return users.find(user => user.id === id);

    },

    users: (parent, args, context, info) => {

      return users;

    }

  }

};

export default resolvers;
```

В каждом резолвере запроса есть четыре аргумента. В запросе user в качестве аргумента передаётся содержимое поля idзаписи базы данных. Сервер возвращает содержимое подходящей записи. Запрос users не содержит аргументов и всегда возвращает весь массив целиком.

Для тестирования получившегося кода перейдите к localhost:4000.

Следующий код должен вернуть список всех записей db.js:

```
query {
```

```
users {
id
name
email
age
}
}
```

Получить первую запись из базы можно с помощью этого кода:

```
query {
user(id: 1) {
id
name
email
age
}
}
```

Изменения

В GraphQL изменения — способ модифицировать данные на сервере и получить обработанную информацию. Этот процесс можно рассматривать как аналогичный концепции CUD (Create, Update, Delete) в стандарте REST.

Для создания изменения откройте файл `schema.graphql` и добавьте новый тип `mutation`:

```
type Mutation {
```

```
createUser(id: ID!, name: String!, email: String!, age: Int): User!

updateUser(id: ID!, name: String, email: String, age: Int): User!

deleteUser(id: ID!): User!

}
```

В данном случае указано три различных изменения:

- createUser: необходимо передать значение полей id, name, email и age. Функция возвращает запись типа User.
- updateUser: необходимо передать значение поля id, новое значение поля name, email или age. Функция возвращает запись типа User.
- deleteUser: необходимо передать значение поля id. Функция возвращает запись типа User.

Теперь откройте файл resolvers.js и ниже объекта Query создайте новый объект mutation:

```
Mutation: {

  createUser: (parent, { id, name, email, age }, context, info) => {

    const newUser = { id, name, email, age };

    users.push(newUser);

    return newUser;

  },

  updateUser: (parent, { id, name, email, age }, context, info) => {

    let newUser = users.find(user => user.id === id);

    newUser.name = name;

    newUser.email = email;

    newUser.age = age;
```

```
    return newUser;
  },
  deleteUser: (parent, { id }, context, info) => {
    const userIndex = users.findIndex(user => user.id === id);
    if (userIndex === -1) throw new Error("User not found.");
    const deletedUsers = users.splice(userIndex, 1);
    return deletedUsers[0];
  }
 }
```

Полный код файла resolvers.js должен выглядеть так:

```
import { users } from "./db";

const resolvers = {
 Query: {
  user: (parent, { id }, context, info) => {
    return users.find(user => user.id === id);
  },
  users: (parent, args, context, info) => {
    return users;
  }
```

```
  },

  Mutation: {

    createUser: (parent, { id, name, email, age }, context, info) => {

      const newUser = { id, name, email, age };

      users.push(newUser);

      return newUser;

    },

    updateUser: (parent, { id, name, email, age }, context, info) => {

      let newUser = users.find(user => user.id === id);

      newUser.name = name;

      newUser.email = email;

      newUser.age = age;

      return newUser;

    },

    deleteUser: (parent, { id }, context, info) => {

      const userIndex = users.findIndex(user => user.id === id);

      if (userIndex === -1) throw new Error("User not found.");

      const deletedUsers = users.splice(userIndex, 1);
```

```
    return deletedUsers[0];
  }
 }
};

export default resolvers;
```

Сделайте запрос к localhost:4000:

```
mutation {
 createUser(id: 3, name: "Robert", email: "robert@gmail.com", age: 21) {
  id
  name
  email
  age
 }
}
```

Он должен вернуть новую запись типа User. Опробуйте также остальные функции изменения.

Подписки

С помощью подписок поддерживается постоянная связь между клиентами и сервером. Базовая подписка выглядит следующим образом:

```
subscription {
 users {
```

```
  id

  name

  email

  age

 }

}
```
Несмотря на то что этот код выглядит похожим на запрос, работает он несколько иначе. При обновлении данных сервер выполняет определённый в подписке запрос GraphQL и рассылает обновлённые данные клиентам.

Заключение

GarphQL набирает популярность. В рамках опроса State of JavaScript, проведённого среди JS-разработчиков, более половины респондентов указали, что слышали об этой технологии и хотели бы с ней ознакомиться, а пятая часть уже её использует и не намерена отказываться.