

A2.2 GraphQL Testing and Reflection

Jasmine Chu

Section I: Testing

Test 1: testDoctorName_success

It can query a doctor's name by their id correctly.

The screenshot shows the GraphQL Playground interface. On the left, the 'Operation' tab is active, displaying a query:

```
1 query testDoctorName_success($doctor_id: ID!) { ...
2   doctorNameById(doctor_id: $doctor_id),
3 }
```

 Below the query, the 'Variables' tab is active, showing a single variable:

```
1 { "doctor_id": "doctor1" }
```

 On the right, the response is displayed in JSON format:

```
{
  "data": {
    "doctorNameById": "Lucy"
  }
}
```

 The status bar at the top right indicates 'STATUS 200', '19.0ms', and '35B'.

Test 2: testDoctorName_fail

500 Internal Server Error: "The doctor doesn't exist!"

The screenshot shows the GraphQL Playground interface. On the left, the 'Operation' tab is active, displaying a query:

```
1 query testDoctorName_fail($doctor_id: ID!) { ...
2   doctorNameById(doctor_id: $doctor_id)
3 }
4
```

 Below the query, the 'Variables' tab is active, showing a single variable:

```
1 {
2   "doctor_id": "doctor3"
3 }
4
```

 On the right, the response is displayed in JSON format, showing an error:

```
{
  "data": {
    "doctorNameById": null
  },
  "errors": [
    {
      "message": "Unexpected error value: \"The doctor doesn't exist!\"",
      "locations": [
        {
          "line": 2,
          "column": 2
        }
      ],
      "path": [
        "doctorNameById"
      ],
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR",
        "stacktrace": [
          "NonErrorThrown: Unexpected error value: \"The doctor doesn't exist!\"",
          "    at toError (/Users/cxy/Desktop/17625/"
```

 The status bar at the top right indicates 'STATUS 200', '23.0ms', and '1.5KB'.

Test 3: testDoctorClinic_success

It can query a doctor's clinic by their id correctly.

The screenshot shows the GraphQL Playground interface. The top bar has a tab labeled 'testDoctorClin...' and a '+' icon. The 'Operation' tab is active, displaying a query:

```
1 query testDoctorClinic_success($doctor_id: ID) {  
2   doctorClinicById(doctor_id: $doctor_id),  
3 }
```

The query is executed, and the response is shown on the right. The status is 200, and the response time is 22.0ms. The response body is:

```
{  
  "data": {  
    "doctorClinicById": "Nice  
Care Clinic"  
  }  
}
```

Below the query editor, the 'Variables' tab is active, showing a single variable:

```
1 {  
  "doctor_id": "doctor1"  
}
```

The response is in JSON format.

Test 4: testDoctorClinic_fail

500 Internal Server Error: "The doctor doesn't exist!"

The screenshot shows the GraphQL Playground interface. The top bar has a tab labeled 'testDoctorClin...' and a '+' icon. The 'Operation' tab is active, displaying a query:

```
1 query testDoctorClinic_fail($doctor_id: ID) {  
2   doctorClinicById(doctor_id: $doctor_id)  
3 }  
4
```

The query is executed, and the response is shown on the right. The status is 200, and the response time is 25.0ms. The response body is:

```
{  
  "data": {  
    "doctorClinicById": null  
  },  
  "errors": [  
    {  
      "message": "Unexpected  
error value: \"The doctor  
doesn't exist!\"",  
      "locations": [  
        {  
          "line": 2,  
          "column": 2  
        }  
      ],  
      "path": [  
        "doctorClinicById"  
      ],  
      "extensions": {  
        "code":  
          "INTERNAL_SERVER_ERROR",  
        "stacktrace": [  
          "NonErrorThrown:  
Unexpected error value: \"The  
doctor doesn't exist!\"",  
          "    at toError (/Users/cxv/Desktop/17625/
```

Below the query editor, the 'Variables' tab is active, showing a single variable:

```
1 {  
2   "doctor_id": "doctor3"  
3 }  
4
```

The response is in JSON format.

Test 5: testDoctorSpecialty_success

It can query a doctor's specialty by their id correctly.

The screenshot shows the GraphQL Playground interface. The top tab is labeled 'testDoctorSpe...'. The 'Operation' tab is active, displaying the query:

```
1 query testDoctorSpecialty_sucess($doctor_id: ID!) {
2   doctorSpecialtyById(doctor_id: $doctor_id),
3 }
```

The 'Variables' tab is also active, showing the variable:

```
1 { "doctor_id": "doctor1" }
```

The right-hand pane shows the response in JSON format:

```
{
  "data": {
    "doctorSpecialtyById":
      "General"
  }
}
```

At the top right of the right pane, the status is 'STATUS 200', the time is '22.0ms', and the size is '43B'.

Test 6: testDoctorSpecialty_fail

500 Internal Server Error: "The doctor doesn't exist!"

The screenshot shows the GraphQL Playground interface. The top tab is labeled 'testDoctorSpe...'. The 'Operation' tab is active, displaying the query:

```
1 query testDoctorSpecialty_fail($doctor_id: ID!) {
2   doctorSpecialtyById(doctor_id: $doctor_id)
3 }
4
```

The 'Variables' tab is also active, showing the variable:

```
1 {
2   "doctor_id": "doctor3"
3 }
4
```

The right-hand pane shows the response in JSON format, indicating an error:

```
{
  "data": {
    "doctorSpecialtyById": null
  },
  "errors": [
    {
      "message": "Unexpected
error value: \"The doctor
doesn't exist!\"",
      "locations": [
        {
          "line": 2,
          "column": 2
        }
      ],
      "path": [
        "doctorSpecialtyById"
      ],
      "extensions": {
        "code":
          "INTERNAL_SERVER_ERROR",
        "stacktrace": [
          "NonErrorThrown:
Unexpected error value: \"The
doctor doesn't exist!\"",
          "    at toError (/
Users/cvv/Desktop/17625/"
        ]
      }
    }
  ]
}
```

At the top right of the right pane, the status is 'STATUS 200', the time is '21.0ms', and the size is '1.5KB'. A 'Submit Operation (⌘ + Enter)' button is visible above the query editor.

Test 7: testDoctorTimeSlots_success

It can query a doctor's time slots by their id correctly.

The screenshot shows a GraphQL query in the 'Operation' tab: `query testDoctorTimeslots_fail($doctor_id: ID!) { doctorTimeslotsById(doctor_id: $doctor_id), }`. The 'Variables' tab shows `{ "doctor_id": "doctor1" }`. The 'Response' tab shows a successful status 200 with a JSON response: `{ "data": { "doctorTimeslotsById": ["0930", "1130"] } }`. The status bar indicates 11.0ms and 49B.

Test 8: testDoctorTimeSlots_fail

500 Internal Server Error: "The doctor doesn't exist!"

The screenshot shows a GraphQL query in the 'Operation' tab: `query testDoctorTimeslots_fail($doctor_id: ID!) { doctorTimeslotsById(doctor_id: $doctor_id) }`. The 'Variables' tab shows `{ "doctor_id": "doctor3" }`. The 'Response' tab shows a 500 status with a JSON response: `{ "data": { "doctorTimeslotsById": null }, "errors": [{ "message": "Unexpected error value: \"The doctor doesn't exist!\"", "locations": [{ "line": 2, "column": 2 }], "path": ["doctorTimeslotsById"], "extensions": { "code": "INTERNAL_SERVER_ERROR", "stacktrace": [{ "NonErrorThrown": "Unexpected error value: \"The doctor doesn't exist!\"", "at toError (/Users/cxy/Desktop/17625/" }] } }] }`. The status bar indicates 19.0ms and 1.5KB.

Test 9: testAddAppointment_success

It added a new appointment successfully, with patient id, doctor id and time.

The screenshot shows the GraphQL Studio interface for a test named `testAddAppointment_success`. The operation is a mutation that calls `addAppointment` with variables `doctor_id`, `patient_id`, and `time`. The response shows a successful status (200) and a JSON body with the appointment ID.

Operation

```
1 mutation testAddAppointment_success($doctor_id:ID!,  
2   $patient_id:ID, $time:String!) {  
3   | addAppointment(doctor_id: $doctor_id,  
4     patient_id: $patient_id, time: $time),  
5 }
```

Variables

```
1 {  
2   "doctor_id": "doctor1",  
3   "patient_id": "patient2",  
4   "time": "1000"  
5 }
```

Response

```
{  
  "data": {  
    "addAppointment":  
      "28a855a2-e6e9-4e0b-a691-4b4e2201baf3"  
  }  
}
```

STATUS 200 | 45.0ms | 67B

Test 10: testAddAppointment_fail

500 Internal Server Error: "Lack of variables: time/doctor_id/patient_id!"

The screenshot shows the GraphQL Studio interface for a test named `testAddAppointment_fail`. The operation is a mutation that calls `addAppointment` with only the `patient_id` variable. The response shows a 500 Internal Server Error with a message indicating a lack of variables.

Operation

```
1 mutation testAddAppointment_fail($patient_id:ID!) {  
2   | addAppointment(patient_id: $patient_id),  
3 }  
4
```

Variables

```
1 {  
2   "patient_id": "patient2"  
3 }  
4
```

Response

```
{  
  "data": {  
    "addAppointment": null  
  },  
  "errors": [  
    {  
      "message": "Unexpected  
error value: \"Lack of  
variables: time/doctor_id/  
patient_id!\"",  
      "locations": [  
        {  
          "line": 2,  
          "column": 3  
        }  
      ],  
      "path": [  
        "addAppointment"  
      ],  
      "extensions": {  
        "code":  
          "INTERNAL_SERVER_ERROR",  
        "stacktrace": [  
          "NonErrorThrown:  
Unexpected error value: \"Lack  
of variables: time/doctor_id/"
```

STATUS 500 | 25.0ms | 1.4KB

Test 11: testCancelAppointment_success

It canceled an appointment successfully

The screenshot shows the GraphQL Studio interface for a test named `testCancelAppointment_success`. The **Operation** tab is active, displaying the following query:

```
1 mutation testCancelAppointment_success ...
2 {
3   $appointment_id: ID! {
4     cancelAppointment(appointment_id:
      $appointment_id)
5   }
6 }
```

The **Variables** tab shows the input variables:

```
1 {
2   "appointment_id": "appointment1"
3 }
```

The **Response** tab shows the JSON output:

```
{
  "data": {
    "cancelAppointment":
      "appointment1"
  }
}
```

Metadata at the top right indicates: STATUS 200, 22.0ms, 46B.

Test 12: testCancelAppointment_fail

500 Internal Server Error: "The appointment doesn't exist!"

The screenshot shows the GraphQL Studio interface for a test named `testCancelAppointment_success`. The **Operation** tab is active, displaying the following query:

```
1 mutation testCancelAppointment_success ...
2 {
3   $appointment_id: ID! {
4     cancelAppointment(appointment_id:
      $appointment_id)
5   }
6 }
```

The **Variables** tab shows the input variables:

```
1 {
2   "appointment_id": "appointment100"
3 }
```

The **Response** tab shows the JSON output, which includes an error:

```
{
  "data": {
    "cancelAppointment": null
  },
  "errors": [
    {
      "message": "Unexpected
        error value: \"The appointment
        doesn't exist!\"",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "cancelAppointment"
      ],
      "extensions": {
        "code":
          "INTERNAL_SERVER_ERROR",
        "stacktrace": [
          "NonErrorThrown:
            Unexpected error value: \"The
            appointment doesn't exist!\"",
            "    at toError (/
              Users/cxy/Desktop/17625/"
          ]
        }
      }
    }
  ]
}
```

Metadata at the top right indicates: STATUS 200, 24.0ms, 1.4KB.

Test 13: testUpdatePatient_success

It updated the patient name for the appointment successfully

Operation

testUpdatePatient_success

```
1 mutation testUpdatePatient_success
2 {
3   updatePatient(appointment_id: $appointment_id,
4     name: $name)
5 }
```

Variables

Headers

JSON

```
1 {
2   "appointment_id": "appointment1",
3   "name": "Lily"
4 }
```

STATUS 200 | 23.0ms | 42B

```
{
  "data": {
    "updatePatient": {
      "appointment1"
    }
  }
}
```

Test 14: testUpdatePatient_fail

500 Internal Server Error: "The appointment doesn't exist!"

testUpdatePat... × +

Operation

testUpdatePatient_fail

```
1 mutation testUpdatePatient_fail ($appointment_id: ID!, $name: String!) {
2   updatePatient(appointment_id: $appointment_id,
3     name: $name)
4 }
```

Variables

Headers

JSON

```
1 {
2   "appointment_id": "appointment3",
3   "name": "Lily"
4 }
```

STATUS 200 | 27.0ms | 1.4KB

```
{
  "data": {
    "updatePatient": null
  },
  "errors": [
    {
      "message": "Unexpected
error value: \"The appointment
doesn't exist!\"",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "updatePatient"
      ],
      "extensions": {
        "code":
"INTERNAL_SERVER_ERROR",
        "stacktrace": [
          "NonErrorThrown:
Unexpected error value: \"The
appointment doesn't exist!\"",
          "    at toError (/
Users/cxy/Desktop/17625/"
        ]
      }
    }
  ]
}
```

Section II: Reflection

- **What were some of the alternative schema and query design options you considered? Why did you choose the selected options?**

For the schema Doctor, Patient, I considered making nearly all variables mandatory and NOT NULL. For the schema Appointment, I only make id NOT NULL to make it easier for users to create an appointment. The old schema I designed is as follows.

```
type Doctor {
  doctor_id: ID!
  name: String!
  clinic: String!
  Specialty: String!
  timeslots: [String]
  appointments: [Appointment] @relation
}

type Patient {
  patient_id: ID!
  name: String!
  appointments: [Appointment]
}

type Appointment {
  appointment_id: ID!
  time: String
  doctor_id: ID
  patient_id: ID
}
```

However, I decided to only make the id NOT NULL. Because it will be easier for clients to create a doctor or patient first. And then use mutations to update detailed information. It is still meaningful to only have doctor_id in a doctor, or only have patient_id in Patient. There are a lot of other variables like name, specialty, clinic and time slots. It may waste time to initiate them or test them. But for the Schema Appointment, I make all variables NOT NULL. Because it is meaningless to have an appointment without patient info/doctor info/time. The final version looks as follows.

```
type Doctor {
  doctor_id: ID!
  name: String
  clinic: String
  Specialty: String
  timeslots: [String]
  appointments: [Appointment] @relation
}
```



```

type Patient {
  patient_id: ID!
  name: String
  appointments: [Appointment]
}
type Appointment {
  appointment_id: ID!
  time: String!
  doctor_id: ID!
  patient_id: ID!
}

```

Another design I have considered is, mutations add and update return Object, delete return Boolean. This design can return more detailed info for clients. Clients can know if the appointment they added or updated has correct info. They can also know if the delete operation has been completed successfully. The old schema I designed is as follows.

```

type Mutation {
  addAppointment(doctor_id: ID, patient_id: ID, time: String): ID
  cancelAppointment(appointment_id: ID): ID
  updatePatient(appointment_id: ID, name: String): ID
}

```

However, I changed this design. I make mutations return the id of the operated appointment. The server will check if the appointment has been added, updated or canceled successfully on the server side. If there is something wrong, the server will throw up errors to remind users. In this way, the users do not have to worry about checking the correctness of the result. Returning only ids can help clients check if they operated the correct appointment more efficiently. The final version looks as follows.

- **Consider the case where, in future, the ‘Event’ structure is changed to have more fields e.g reference to patient details, consultation type (first time/follow-up etc.) and others.**

- **What changes will the clients (API consumer) need to make to their existing queries (if any).**

No changes needed to be made.

- **How will you accommodate the changes in your existing Schema and Query types?**

I will add an Object Patient in the Type Appointment(Event), then add other variables like consultation_type.

```
type Appointment {  
    appointment_id: ID!  
    time: String!  
    doctor_id: ID!  
    patient_id: ID!  
    patient: Patient!  
    consultation_type: String!  
    ...  
}
```

I will also change the resolvers as follows to connect Appointment with Patient info.

```
const Appointment = {  
    patient: (appointment) => db.patients.filter(p => p.patient_id ===  
    appointment.patient_id),  
}
```

- **Describe two GraphQL best practices that you have incorporated in your API design.**

One of the best practices I use is versionless. As we discussed on the last question, if adding fields in the existing schema, clients do not need to make any changes. Clients are not affected as long as existing fields are not removed from the schema. Clients can only ask what they need. Versionless makes my GraphQL schema easy to maintain, extend and evolve.

Another best practice I use is the naming of the queries. I avoided containing “get” in the query name. For example, I use “doctorNameById” instead of “getDoctorNameById”. Besides, each of my queries do just one thing. All of them are self-explanation.