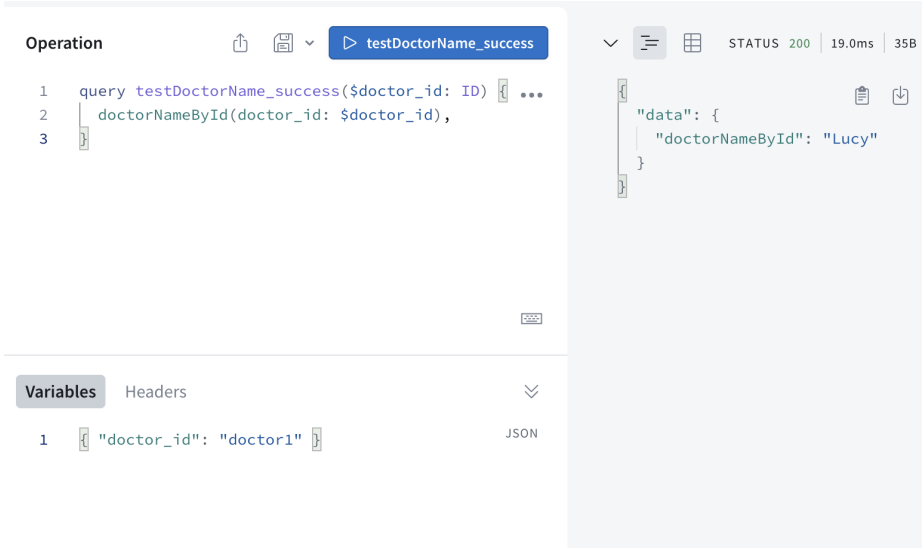
A2.2 GraphQL Testing and Reflection

Jasmine Chu

Section I: Testing

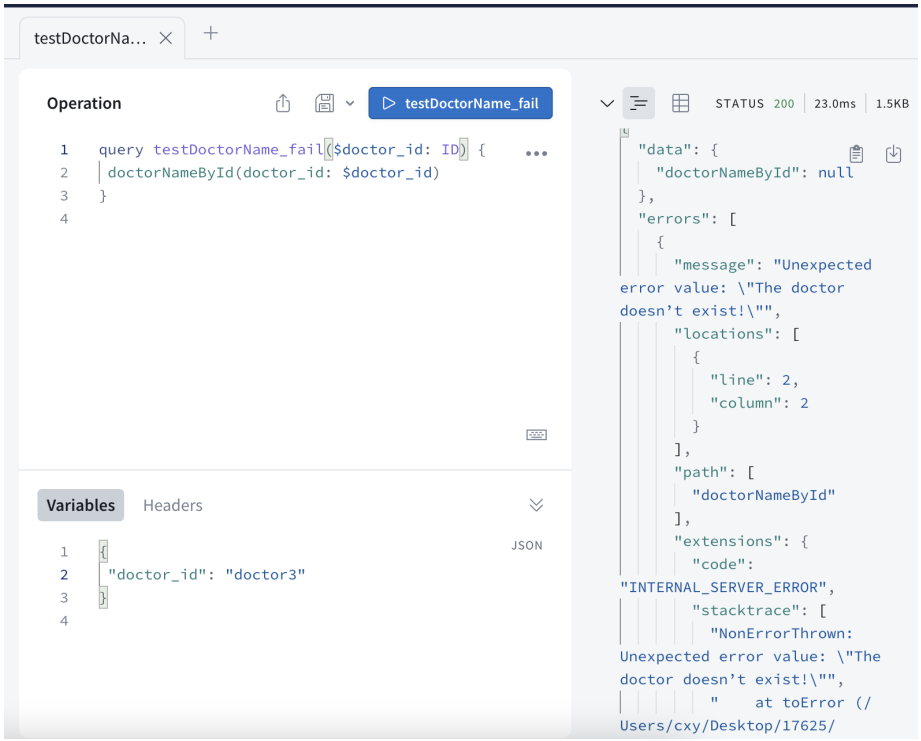
Test 1: testDoctorName_success

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



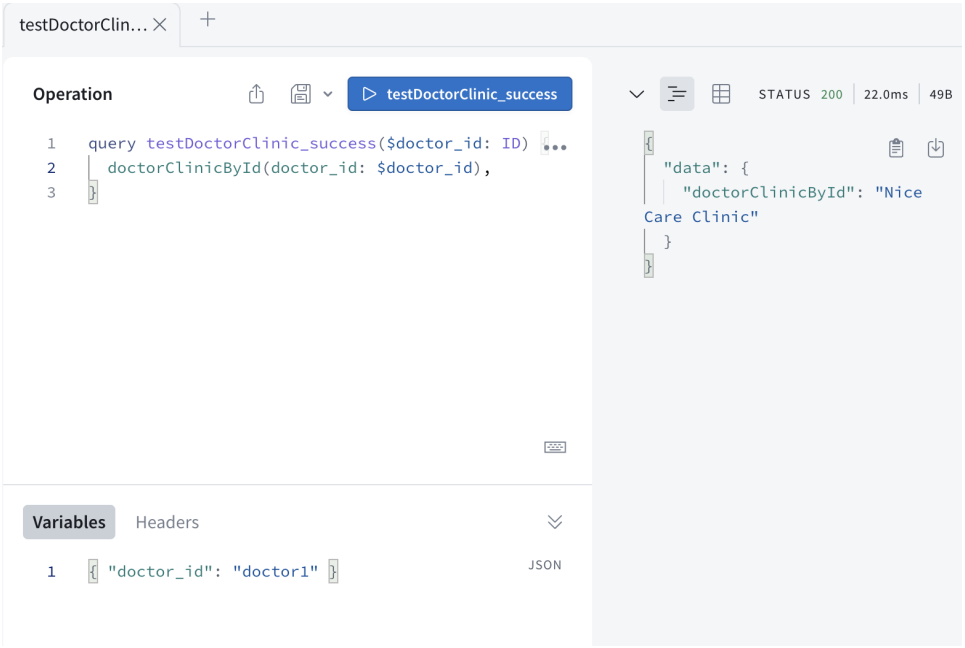
Test 2: testDoctorName_fail

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



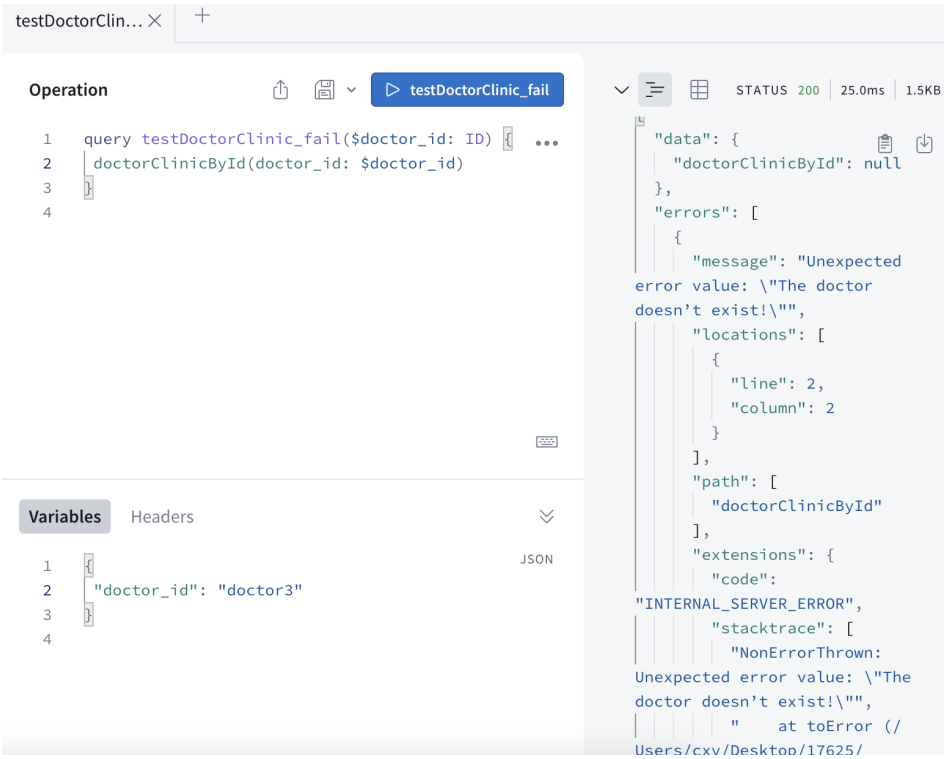
Test 3: testDoctorClinic_success

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



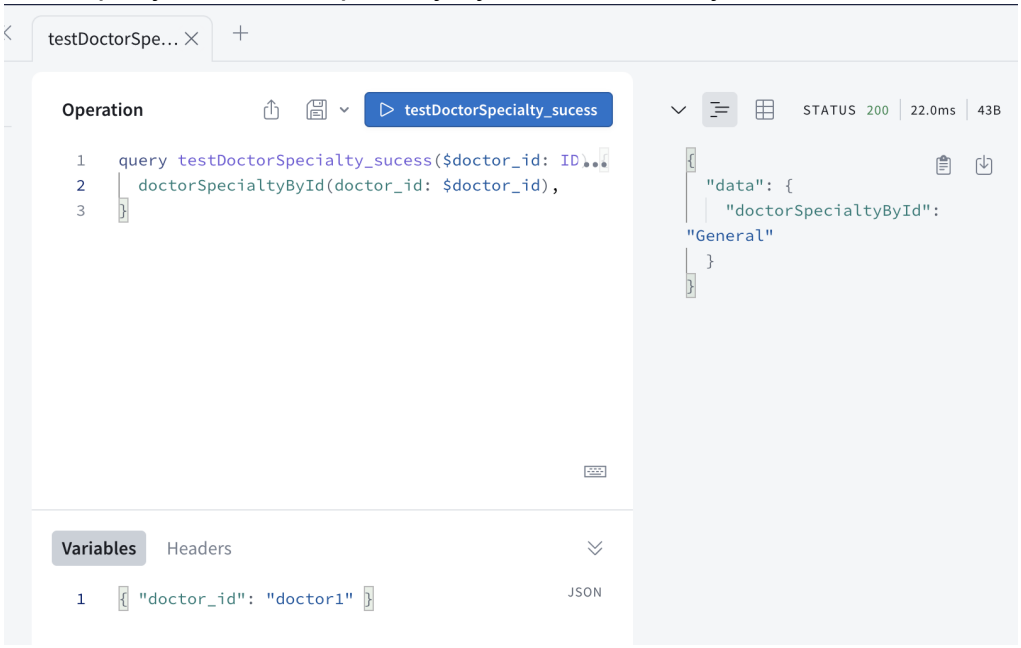
Test 4: testDoctorClinic_fail

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



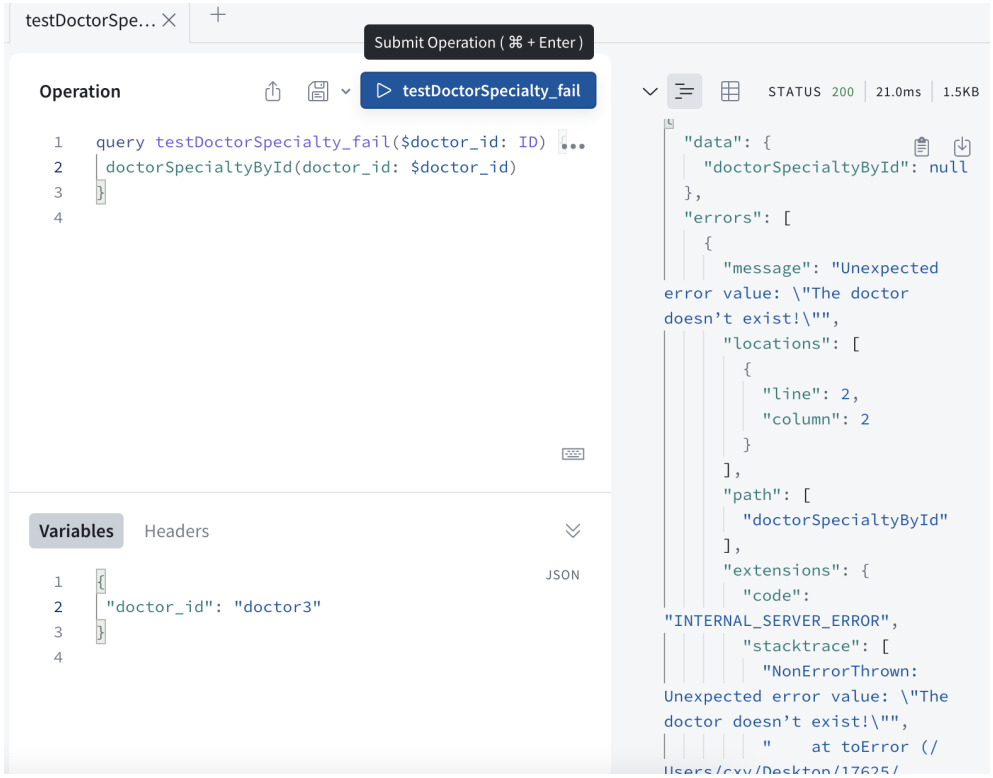
Test 5: testDoctorSpecialty_success

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



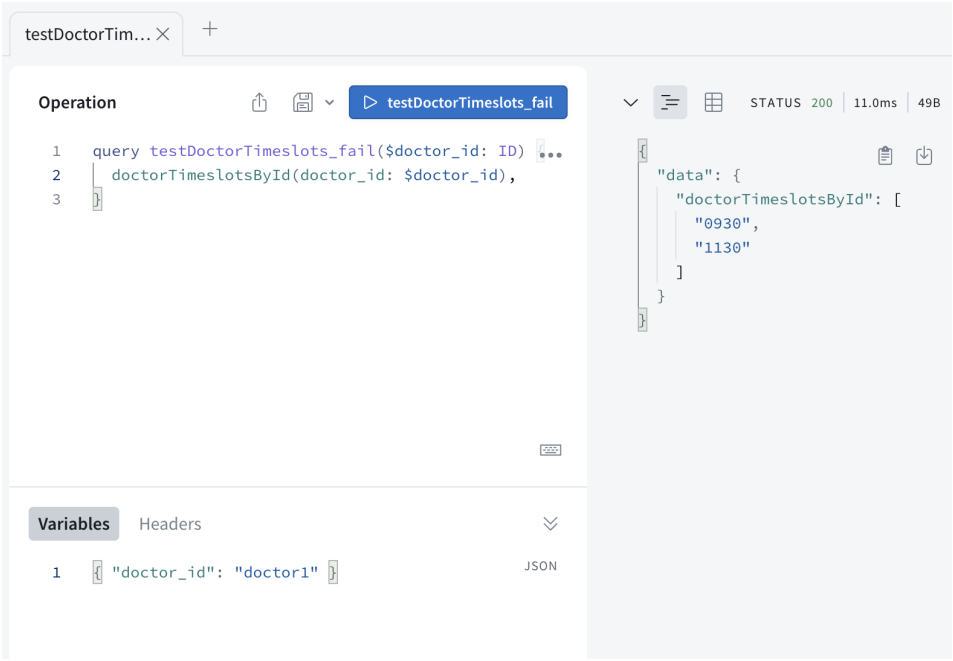
Test 6: testDoctorSpecialty_fail

500 Internal Server Error: “The doctor doesn’t exist!”



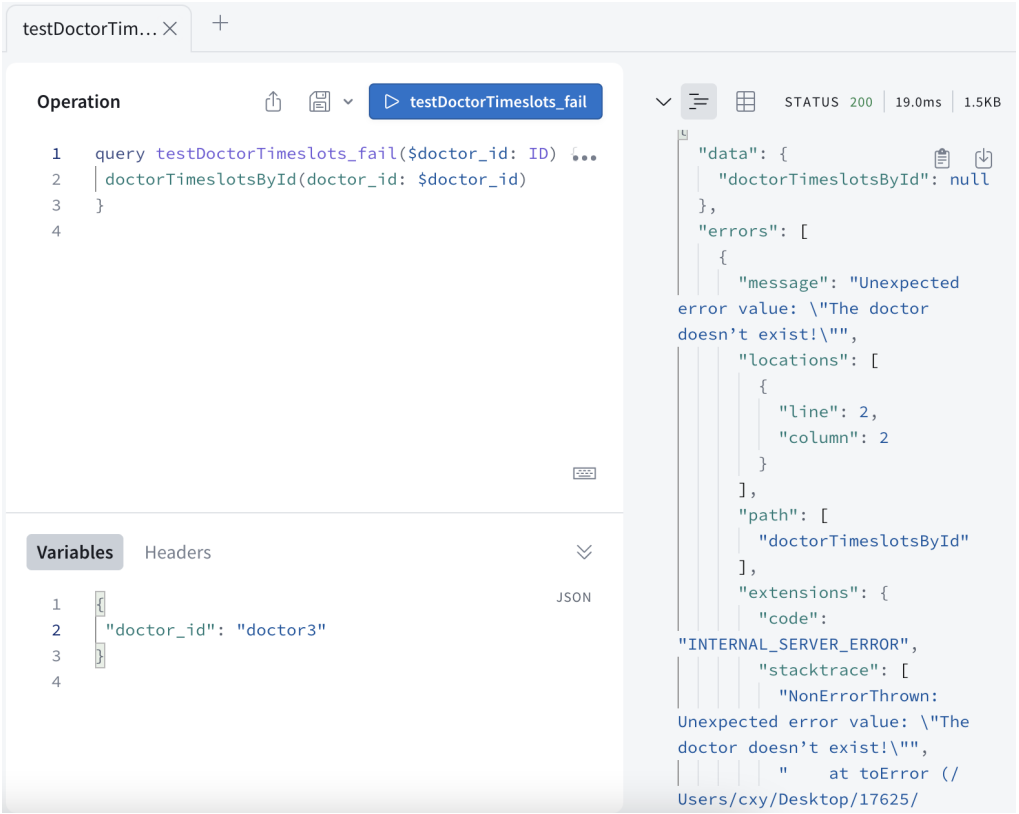
Test 7: testDoctorTimeSlots_success

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



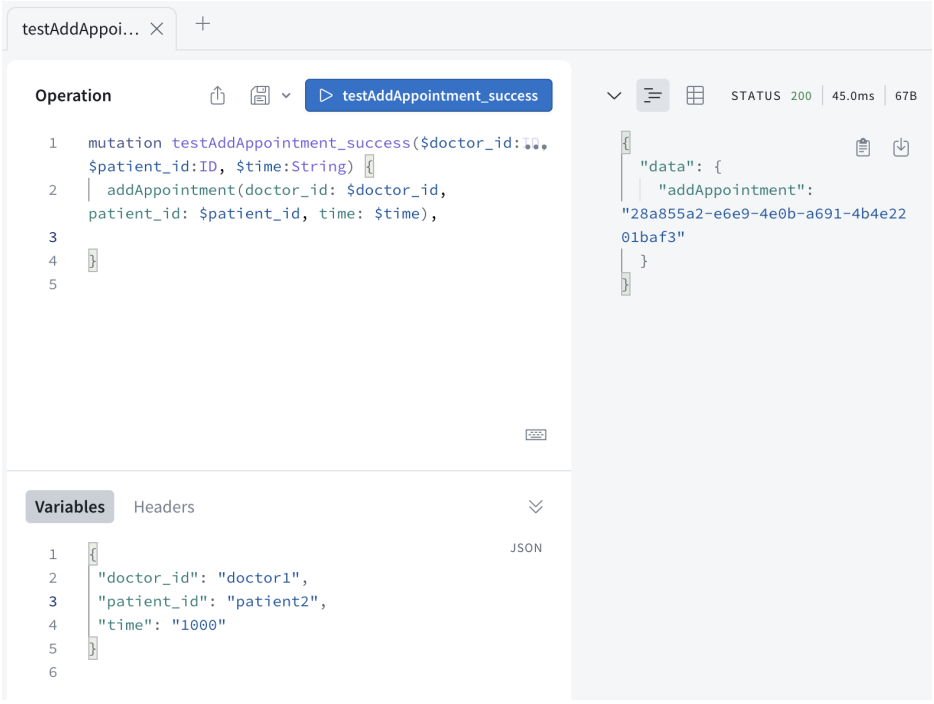
Test 8: testDoctorTimeSlots_fail

500 Internal Server Error: “The doctor doesn’t exist!”



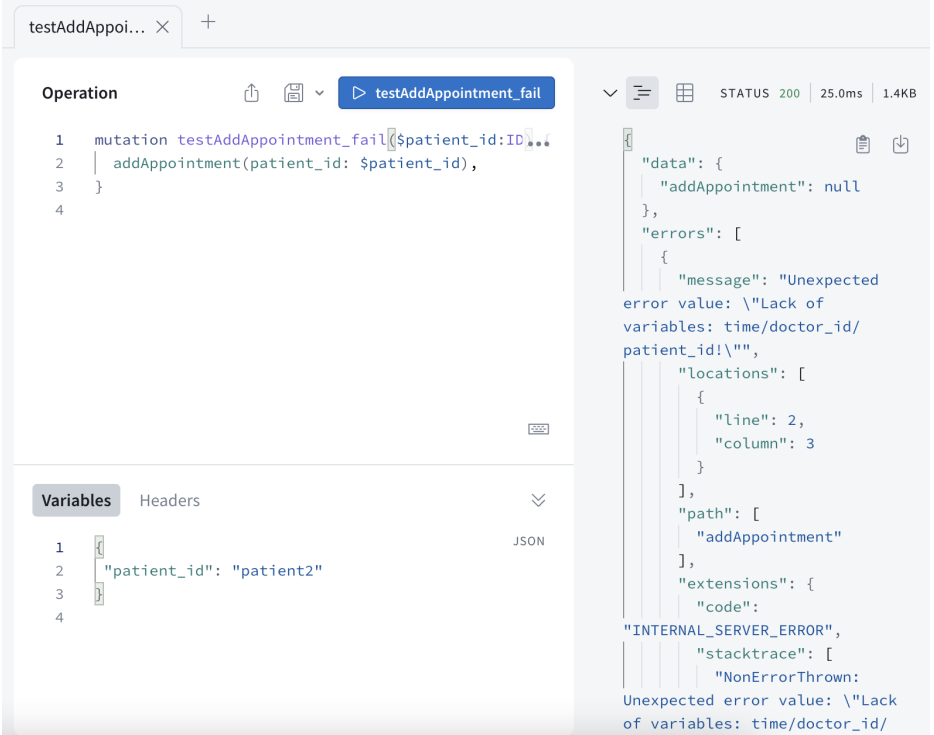
Test 9: testAddAppointment_success

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



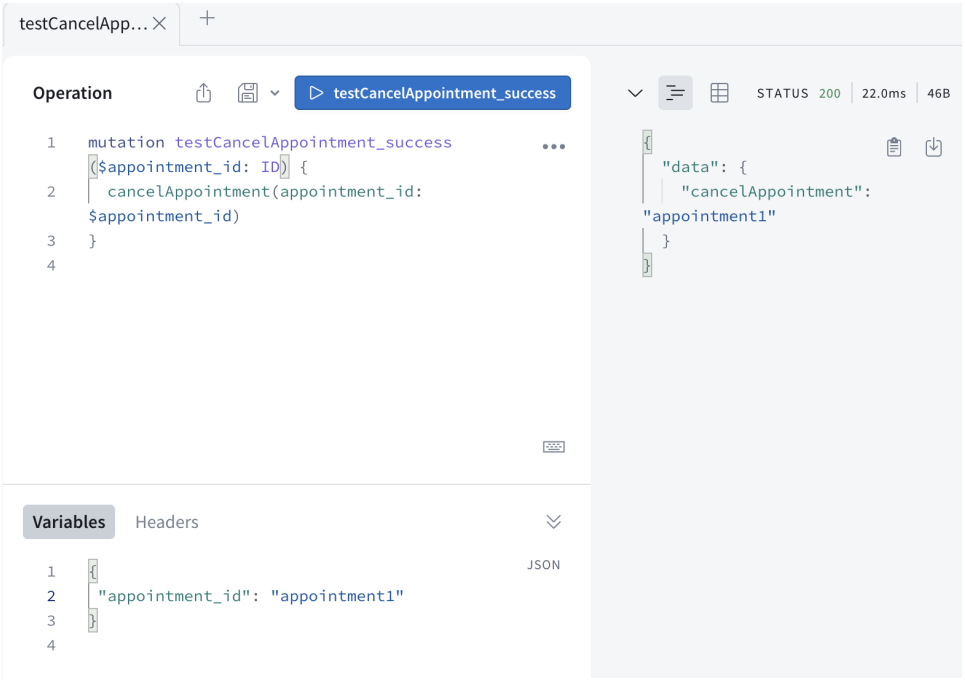
Test 10: testAddAppointment_fail

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



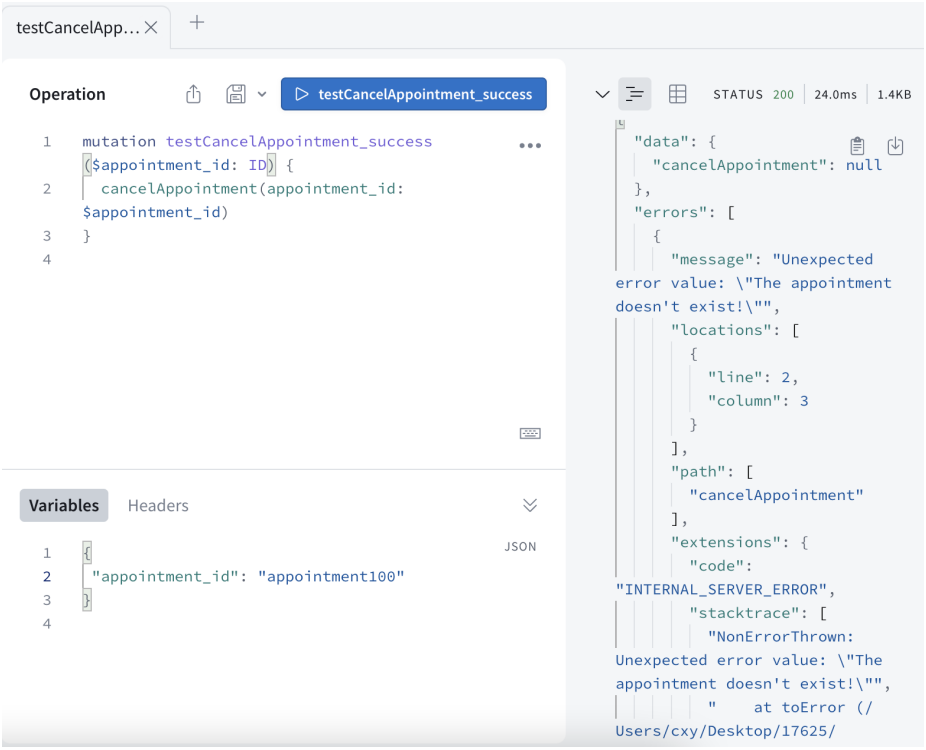
Test 11: testCancelAppointment_success

It canceled an appointment successfully



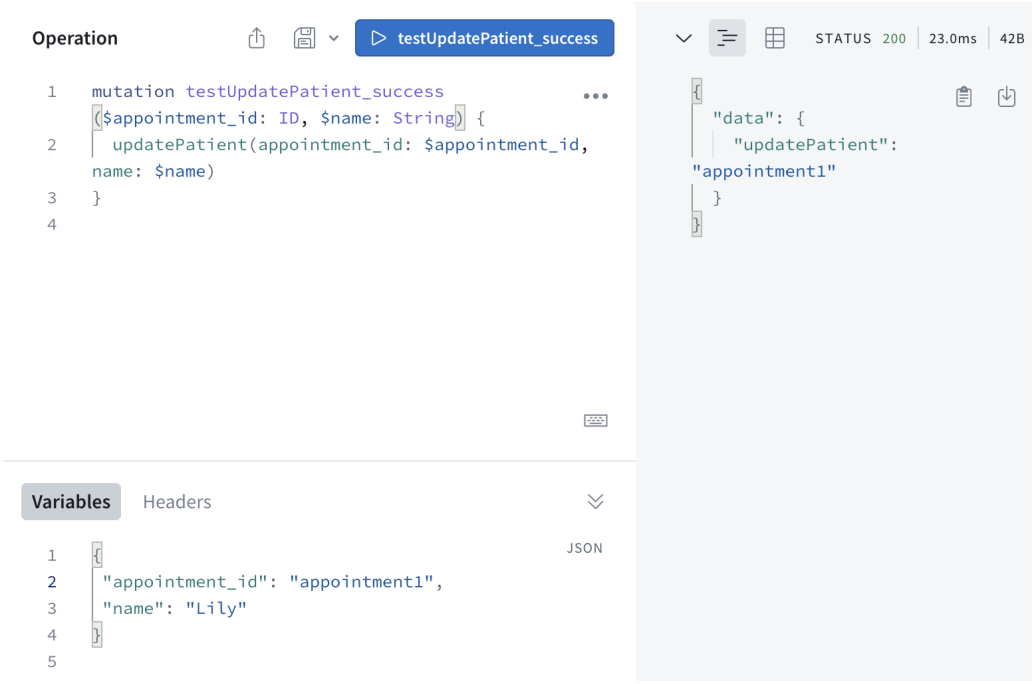
Test 12: testCancelAppointment_fail

500 Internal Server Error: “The appointment doesn’t exist!”



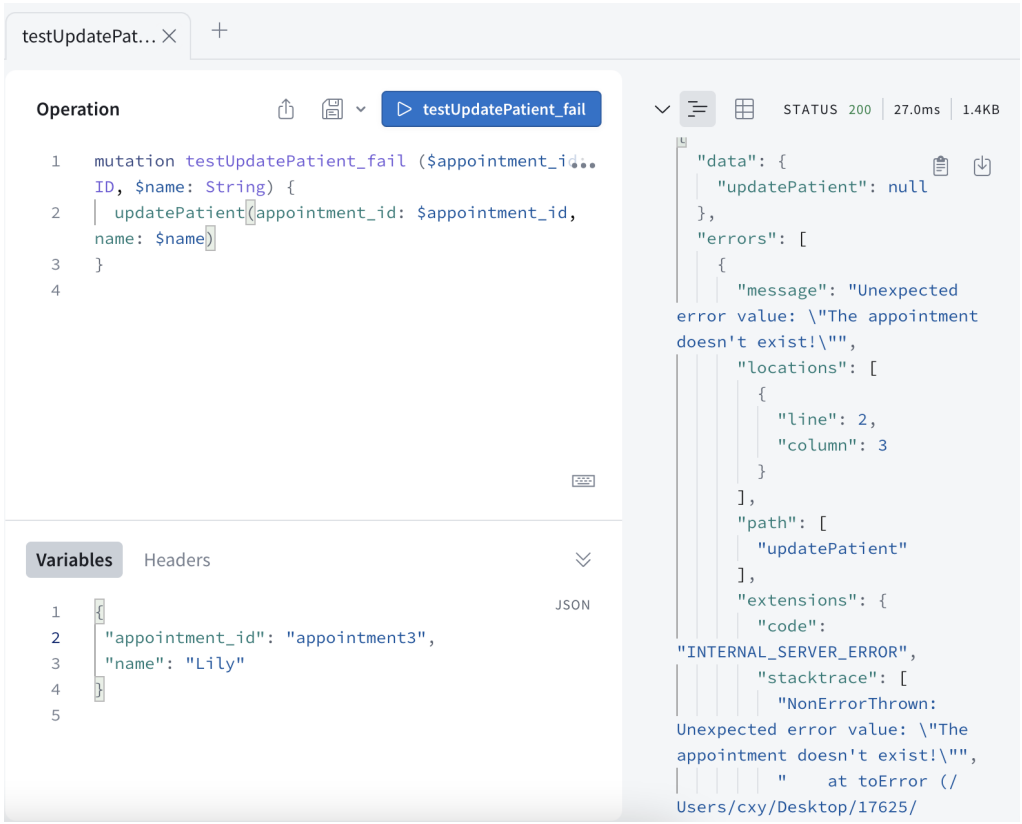
Test 13: testUpdatePatient_success

It updated the patient name for the appointment successfully



Test 14: testUpdatePatient_fail

500 Internal Server Error: “The appointment doesn’t exist!”



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 evolute.

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.