

API Design Assignment A2.2

Testing

Notes:

The test case identifiers follow the following format: [Q/M][int].[0/1]. Q/M stands for Query/Mutation. The following int is the serial number, and the binary digit after the dot stands for happy path test case (1) and error condition test case (0).

This API is for the use of the developers to perform CRUD operations.

Happy path test cases for the queries are mentioned below.

Q1.1. getDoctorDetailsHappy

Here the input is a valid doctor_id. The expected output is the doctor_name, clinic, and specialty.

The screenshot displays a GraphQL sandbox interface with the following components:

- Documentation:** Shows the query `getDoctorDetails` with the argument `doctor_id: ID!` and fields `clinic_name: String`, `doctor_name: String`, and `specialty: String`.
- Operation:** A GraphQL query is defined:

```
1 query ExampleQuery($doctorId: ID!) {  
2   getDoctorDetails(doctor_id: $doctorId) {  
3     clinic_name  
4     doctor_name  
5     specialty  
6   }  
7 }
```
- Variables:** A JSON object is provided:

```
1 {  
2   "doctorId": 1  
3 }
```
- Response:** The resulting JSON response is shown:

```
{  
  "data": {  
    "getDoctorDetails": {  
      "clinic_name": "clinic1",  
      "doctor_name": "alpha",  
      "specialty": "general_physician"  
    }  
  }  
}
```
- Status:** The response status is 200, with a time of 29.0ms and a size of 110B.

Q2.1. getDoctorSlotsHappy

Here the input is a valid doctor_id. The expected output is the array of available_timeslots.

The screenshot displays a GraphQL sandbox interface with the following components:

- Header:** Includes a "SANDBOX" label, a URL "http://localhost:4000/", a "Publish" button, and a "Log in" button.
- Left Panel (Documentation):**
 - Shows the path "Root > Query > getDoctorSlots".
 - Displays the field "getDoctorSlots: [Int]" with a checkmark.
 - Arguments:** Lists "doctor_id: ID!".
 - Metadata for Int Type:** States "The Int scalar type represents non-fractional signed whole numeric values. Int can represent values between -2^{31} and $2^{31} - 1$."
- Operation Panel:**
 - Contains the query:

```
1 query ExampleQuery($doctorId: ID!) {  
2   getDoctorSlots(doctor_id: $doctorId)  
3 }
```
 - Variables:** Shows a JSON object:

```
1 {  
2   "doctorId": 1  
3 }
```
- Response Panel:**
 - Shows the JSON response:

```
{  
  "data": {  
    "getDoctorSlots": [  
      1,  
      2,  
      3,  
      4,  
      5,  
      6,  
      7,  
      8,  
      9,  
      10,  
      11,  
      12,  
      13,  
      14,  
      15,  
      16  
    ]  
  }  
}
```
 - Metadata: "STATUS 200 | 49.0ms | 69B"

Happy path test cases for the mutations are mentioned below.

M1.1. bookAppointmentHappy

Here the input is a valid `doctor_id`, a valid `patient_id`, and an `appointment_timeslot` which is present in the `available_timeslots` array. The expected output is the `doctor_name`, the new `available_timeslots` array from which the `appointment_timeslot` has been removed, `patient_name`, and `appointment_timeslot`.

The screenshot displays the GraphQL Playground interface with a successful mutation query. The left sidebar shows the documentation for the `bookAppointment` mutation, listing its arguments and fields. The main area shows the query, variables, and the resulting JSON response.

Documentation:

- Root > Mutation > bookAppointment
- bookAppointment: AppointmentConfirmation
- Arguments:
 - doctor_id: ID!
 - patient_id: ID!
 - appointment_timeslot: Int
- Fields:
 - appointment_timeslot: Int
 - available_timeslots: [Int]
 - doctor_name: String
 - patient_name: String

Operation:

```
1 mutation Mutation($doctorId: ID!, $patientId: ID!, $appointmentTimeslot: ...  
2 Int) {  
3   bookAppointment(doctor_id: $doctorId, patient_id: $patientId,  
4     appointment_timeslot: $appointmentTimeslot) {  
5     appointment_timeslot  
6     available_timeslots  
7     doctor_name  
8     patient_name  
9   }  
10 }
```

Variables:

```
1 {  
2   "doctorId": 1,  
3   "patientId": 100,  
4   "appointmentTimeslot": 5  
5 }
```

Response:

```
{  
  "data": {  
    "bookAppointment": {  
      "appointment_timeslot": 5,  
      "available_timeslots": [  
        1,  
        2,  
        3,  
        4,  
        6,  
        7,  
        8,  
        9,  
        10,  
        11,  
        12,  
        13,  
        14,  
        15,  
        16  
      ],  
      "doctor_name": "alpha",  
      "patient_name": "alice"  
    }  
  }  
}
```

STATUS: 200 | 35.0ms | 162B

M2.1. cancelAppointmentHappy

Here the input is a valid appointment_id. The expected output is the doctor_id, and the new available_timeslots array to which the appointment_timeslot has been added back.

The screenshot displays a GraphQL IDE interface with a dark theme. The top bar shows a 'SANDBOX' environment at 'http://localhost:4000/' with a 'Publish' button and a 'Log In' link. The left sidebar contains a 'Documentation' panel for the 'cancelAppointment' operation, showing its arguments and fields. The main area is divided into three sections: 'Operation', 'Variables', and 'Response'.

Operation: A GraphQL mutation query is shown, canceling an appointment and returning the doctor's ID and the updated available timeslots.

```
1 mutation Mutation($appointmentId: ID!) {  
2   cancelAppointment(appointment_id: $appointmentId) {  
3     available_timeslots  
4     doctor_id  
5   }  
6 }
```

Variables: The input variables for the mutation are defined in a JSON object.

```
1 {  
2   "appointmentId": "1000"  
3 }
```

Response: The JSON response from the server, showing the successful cancellation and the updated available timeslots.

```
{  
  "data": {  
    "cancelAppointment": {  
      "available_timeslots": [  
        1,  
        3,  
        4,  
        5,  
        6,  
        7,  
        8,  
        9,  
        10,  
        11,  
        12,  
        13,  
        14,  
        15,  
        16,  
        2  
      ],  
      "doctor_id": "2"  
    }  
  }  
}
```

The status bar at the top right indicates a successful response with a status of 200, a response time of 57.0ms, and a size of 112B.

M3.1. updatePatientNameHappy

Here the input is a valid appointment_id, and the new patient_name. The expected output is a new patient_id for this new patient, the new patient_name, and the appointment_timeslot.

The screenshot displays a GraphQL Sandbox interface with the following components:

- Header:** Includes a 'SANDBOX' tab, a URL 'http://localhost:4000/', a 'Publish' button, and a 'Login' button.
- Left Panel (Documentation):** Shows the breadcrumb 'Root > Mutation > updatePatientName'. The operation 'updatePatientName: NewPatientAppointment' is selected. It lists arguments: 'appointment_id: ID!' and 'patient_name: String'. It also lists fields: 'appointment_timeslot: Int!', 'patient_id: ID!', and 'patient_name: String'.
- Operation Panel:** Contains the GraphQL query:

```
1 mutation Mutation($appointmentId: ID!, $patientName: String) {  
2   updatePatientName(appointment_id: $appointmentId, patient_name:  
3     $patientName)  
4     {  
5       appointment_timeslot  
6       patient_id  
7       patient_name  
8     }  
9 }
```
- Response Panel:** Shows the JSON response:

```
{  
  "data": {  
    "updatePatientName": {  
      "appointment_timeslot": 2,  
      "patient_id": "300",  
      "patient_name": "vaishnavi"  
    }  
  }  
}
```

 The status is '200' and the response size is '104B'.
- Variables Panel:** Shows the input variables:

```
1 {  
2   "appointmentId": "1000",  
3   "patientName": "vaishnavi"  
4 }
```

Error condition test cases for the queries are mentioned below.

Q1.0. getDoctorDetailsUnhappy

Here the input is an invalid doctor_id. The expected output is an error.

The screenshot shows a GraphQL IDE interface with the following components:

- Documentation:** Shows the schema for `getDoctorDetails: DoctorDetails`. Arguments include `doctor_id: ID!`. Fields include `clinic_name: String`, `doctor_name: String`, and `specialty: String`.
- Operation:** Contains the query:

```
1 query Query($doctorId: ID!) {  
2   getDoctorDetails(doctor_id: $doctorId) {  
3     clinic_name  
4     doctor_name  
5     specialty  
6   }  
7 }
```
- Variables:** Shows the variable `doctorId` with the value `13`.

```
1 {  
2   "doctorId": 13  
3 }
```
- Response:** Shows the JSON response:

```
{  
  "data": {  
    "getDoctorDetails": null  
  }  
}
```
- Status:** The response status is 200, with a response time of 35.0ms and a size of 35B.

Q2.0. getDoctorSlotsUnhappy

Here the input is an invalid doctor_id. The expected output is an error.

The screenshot shows the GraphQL Playground interface. On the left, the 'Documentation' panel for the `getDoctorSlots` query is visible, showing its arguments and metadata. The 'Query' tab is active, displaying the following query:

```
1 query Query($doctorId: ID!) {  
2   | getDoctorSlots(doctor_id: $doctorId)  
3 }
```

The 'Variables' section shows the input:

```
1 {  
2   "doctorId": 13  
3 }
```

The 'Response' panel shows the JSON output, which includes an error:

```
{  
  "data": {  
    "getDoctorSlots": null  
  },  
  "errors": [  
    {  
      "message": "Cannot read properties of undefined (reading  
'available_timeslots')",  
      "locations": [  
        {  
          "line": 2,  
          "column": 3  
        }  
      ],  
      "path": [  
        "getDoctorSlots"  
      ],  
      "extensions": {  
        "code": "INTERNAL_SERVER_ERROR",  
        "stacktrace": [  
          "TypeError: Cannot read properties of undefined (reading  
'available_timeslots')",  
            "    at Object.getDoctorSlots (file:///Users/vaishnavi/  
graphql-assignment/index.js:132:21)",  
            "    at field.resolve (file:///Users/vaishnavi/graphql-assignment/  
node_modules/@apollo/server/dist/esm/utils/schemaInstrumentation.js:36:28)",  
            "    at executeField (/Users/vaishnavi/graphql-assignment/  
node_modules/graphql/execution/execute.js:481:20)",  
            "    at executeFields (/Users/vaishnavi/graphql-assignment/  
node_modules/graphql/execution/execute.js:413:20)",  
            "    at executeOperation (/Users/vaishnavi/graphql-assignment/  
node_modules/graphql/execution/execute.js:374:20)"  
          ]  
        }  
      ]  
    }  
  ]  
}
```

The status bar at the top right indicates a 200 status code, 45.0ms execution time, and 1.5KB response size.

Error condition test cases for the mutations are mentioned below.

M1.0. bookAppointmentUnhappy

Here the input is an invalid `doctor_id`, an invalid `patient_id`, and an `appointment_timeslot` which is not present in the `available_timeslots` array. The expected output is an error.

The screenshot displays a GraphQL Sandbox interface with the following components:

- Documentation:** Shows the schema for the `bookAppointment` mutation. Arguments include `doctor_id: ID!`, `patient_id: ID!`, and `appointment_timeslot: Int`. Fields include `appointment_timeslot: Int`, `available_timeslots: [Int]`, `doctor_name: String`, and `patient_name: String`.
- Operation:** A GraphQL mutation query is entered:

```
1 mutation Mutation($doctorId: ID!, ...  
2 $patientId: ID!, $appointmentTimeslot:  
3 Int) {  
4   bookAppointment(doctor_id: $doctorId,  
5   patient_id: $patientId,  
6   appointment_timeslot:  
7     $appointmentTimeslot)  
8   {  
9     appointment_timeslot  
10    available_timeslots  
11    doctor_name  
12    patient_name  
13  }  
14 }
```
- Variables:** A JSON object is provided:

```
1 {  
2   "doctorId": 13,  
3   "patientId": 26,  
4   "appointmentTimeslot": 50  
5 }
```
- Response:** The status is 200, and the response is a JSON object indicating an error:

```
{  
  "data": {  
    "bookAppointment": null  
  },  
  "errors": [  
    {  
      "message": "Cannot read properties of undefined (reading  
'available_timeslots')",  
      "locations": [  
        {  
          "line": 2,  
          "column": 3  
        }  
      ],  
      "path": [  
        "bookAppointment"  
      ],  
      "extensions": {  
        "code": "INTERNAL_SERVER_ERROR",  
        "stacktrace": [  
          "TypeError: Cannot read properties of undefined (reading  
'available_timeslots')",  
          "    at Object.bookAppointment (file:///Users/vaishnavi/  
graphql-assignment/index.js:141:50)",  
          "    at field.resolve (file:///Users/vaishnavi/graphql-assignment/  
node_modules/@apollo/server/dist/esm/utils/schemaInstrumentation.js:36:28)",  
          "    at executeField (/Users/vaishnavi/graphql-assignment/  
node_modules/graphql/execution/execute.js:481:20)",  
          "    at /Users/vaishnavi/graphql-assignment/node_modules/graphql/  
execution/execute.js:377:22",  
          "    at promiseReduce (/Users/vaishnavi/graphql-assignment/  
node_modules/graphql/execution/execute.js:138:24)"  
        ]  
      }  
    }  
  ]  
}
```


M2.0. cancelAppointmentUnhappy

Here the input is an invalid appointment_id. The expected output is an error.

The screenshot shows the GraphQL Playground interface. On the left, the 'Documentation' panel displays the 'cancelAppointment' mutation under the 'Mutation' type. The arguments are 'appointment_id: ID!'. The fields are 'available_timeslots: [Int]' and 'doctor_id: ID!'. The 'Operation' tab shows the following query:

```
1 mutation Mutation($appointmentId: ID!) ...
2   cancelAppointment(appointment_id:
3     $appointmentId) {
4     available_timeslots
5     doctor_id
6   }
```

The 'Variables' tab shows the input variables:

```
1 {
2   "appointmentId": 5
3 }
```

The 'Response' tab shows the JSON response:

```
{
  "data": {
    "cancelAppointment": null
  },
  "errors": [
    {
      "message": "Cannot read properties of undefined (reading 'doctor_id')",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "cancelAppointment"
      ],
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR",
        "stacktrace": [
          "TypeError: Cannot read properties of undefined (reading 'doctor_id')",
          "    at Object.cancelAppointment (file:///Users/vaishnavi/graphql-assignment/index.js:158:36)",
          "    at field.resolve (file:///Users/vaishnavi/graphql-assignment/node_modules/@apollo/server/dist/esm/utils/schemaInstrumentation.js:36:28)",
          "    at executeField (/Users/vaishnavi/graphql-assignment/node_modules/graphql/execution/execute.js:481:20)",
          "    at /Users/vaishnavi/graphql-assignment/node_modules/graphql/execution/execute.js:377:22",
          "    at promiseReduce (/Users/vaishnavi/graphql-assignment/"
```

M3.0. updatePatientNameUnhappy

Here the input is an invalid appointment_id, and the new patient_name. The expected output is an error.

The screenshot shows the GraphQL Playground interface. On the left, the 'Documentation' panel displays the 'updatePatientName' mutation under the 'Mutation' type. The arguments are 'appointment_id: ID!' and 'patient_name: String'. The fields are 'appointment_timeslot: Int', 'patient_id: ID!', and 'patient_name: String'. The 'Operation' tab shows the following query:

```
1 mutation Mutation($appointmentId: ID!, ...
2   $patientName: String) {
3     updatePatientName(appointment_id:
4       $appointmentId, patient_name:
5       $patientName) {
6       appointment_timeslot
7       patient_id
8       patient_name
9     }
10  }
```

The 'Variables' tab shows the input variables:

```
1 {
2   "appointmentId": 5,
3   "patientName": "vaishnavi"
4 }
```

The 'Response' tab shows the JSON response:

```
{
  "data": {
    "updatePatientName": null
  },
  "errors": [
    {
      "message": "Cannot read properties of undefined (reading 'patient_id')",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "updatePatientName"
      ],
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR",
        "stacktrace": [
          "TypeError: Cannot read properties of undefined (reading 'patient_id')",
          "    at Object.updatePatientName (file:///Users/vaishnavi/graphql-assignment/index.js:177:37)",
          "    at field.resolve (file:///Users/vaishnavi/graphql-assignment/node_modules/@apollo/server/dist/esm/utils/schemaInstrumentation.js:36:28)",
          "    at executeField (/Users/vaishnavi/graphql-assignment/node_modules/graphql/execution/execute.js:481:20)",
          "    at /Users/vaishnavi/graphql-assignment/node_modules/graphql/execution/execute.js:377:22",
          "    at promiseReduce (/Users/vaishnavi/graphql-assignment/"
```

Reflection

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

Some alternative design options considered were having separate entities or types for clinics and specialties as well. However, I decided to not go with that option because those elements can be efficiently captured in the Doctor entity itself, which is what I have done. Another alternative considered was not having a Patient entity at all, as its information can be conveyed through just the Doctor and Appointment entities. However, I decided to not go with that option either because it might have made the schema harder to understand, and the queries more complex. That is why, I decided that having three entities or types namely the Doctor, Patient, and Appointment is the best option given the mentioned factors.

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).

My design already incorporates the Patient as an object instead of just the patient name, so the only changes required from the client's side would be for them to select which fields they want, and they have additional options now like patient details, consultation type (first time/follow-up etc.).

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

I can accommodate the changes in my existing design by adding these additional patient details and consultation type (first time/follow-up etc.) fields to my type definitions and the Patient and Doctor entities in my original ER diagram.

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

1. I have expressed my schema in an accepted 'language' such as the ER diagram shown in the A2.1 document.
2. I have used ID type for unique identifiers and non-nullable types for mandatory fields.
3. While designing queries, I have use object types instead of object ids.