17625 – API Design
Assignment 2.1: GraphQL API
Shashank Shekhar, sshekha3

Task 1.1: Design Subtask

To design the GraphQL API for the given system, we first need to understand the domain of the problem. After defining the domain, we work on understanding the different resources that exist and how are they related to each other. Based on these resources and their relation we generate the ER or UML diagram. We then define the different type of queries that we should have to support the given system. All the above-mentioned features are discussed as follows:

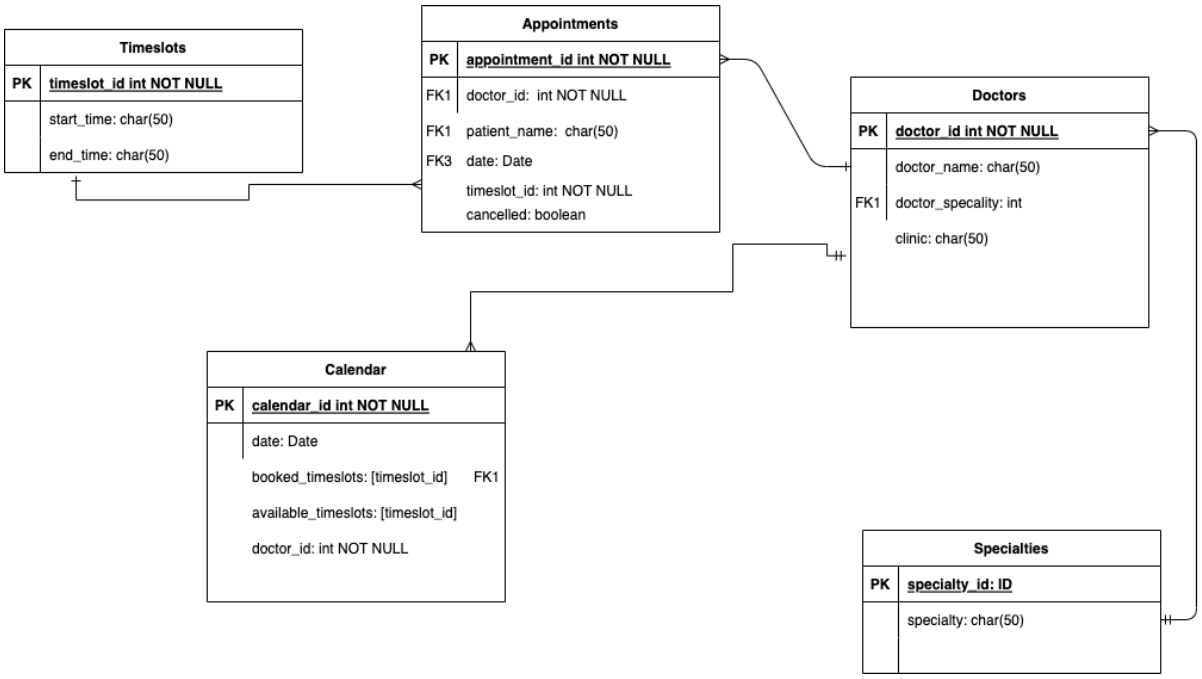
Domain: We are developing an API for *Doctor's Assistant Inc (DAI).* which is a software company that designs platform to support small medical practices. Through their platform, the business can increase their online presence, manage patients and their visits. Recent growth of DAI has necessitated the design of a new API that caters to a more diverse audience.

Resources: In the DAI system, we have the following resources:

- Doctors: The system need to store the information about a doctor and related details such as name, clinic name, specialty, experience, etc.
- Patient: The system needs to store patient details and provide capability to add, modify or remove a patient.
- Appointment Details: The system also needs to manage appointment details for a doctor and patient and provide functionality to book and manage appointments. It should be capable of handling calendar for a given doctor or patient.
- Timeslots: System should also store and manage timeslots for a doctor.
- Specialties: The system should manage different specialties of doctors.

Object graph:
The entity relation diagram for the DAI system is as shown in the below figure (Figure 1). It shows all the database tables maintained to achieve the required system behavior.

Doctors table stores all the information about doctors using DAI. Similarly, Specialties table handles supported specialties, and Timeslots table, supported timeslots. Also, a separate Calendar table is to support the calendar for each doctor and provide information regarding available timeslots. Appointments table handles the individual appointments which correspond to a date, timeslot and doctor and stores the patient information.



Schema:
Based upon above ERD and requirements of the system, following schema types can be defined for the GraphQL API.

Doctor:

```
type Doctor {
       id: ID!
       name: String
       specialty: Specialty
       clinic: String
}
```

Specialty:

```
type Specialty {
       id: ID!
       specialty: String
}
```

Timeslot:

```
type Timeslot {
	id: ID!
	startTime: String
	endTime: String
}
```

Appointment:

```
type Appointment {
	id: ID!
	doctor: Doctor
	patientName: String
	timeslot: Timeslot
	date: String
	cancelled: Boolean
}
```

Calendar:

```
type Calendar {
	id: ID!
	date: String
	doctor: Doctor
	bookedTimeslots: [Timeslot]
	availableTimeslots: [Timeslot]
}
```

Queries: As per the requirements, the API should support following capabilities:

1. Get details for a doctor (name, clinic, specialty).
2. Get doctor's available timeslots for today.
3. Book an appointment with doctor for today.
4. Cancel an appointment
5. Update name of the patient for an appointment.

To support the above capabilities, we need to define the *Query* and *Mutation* our GraphQL API will support. Below, we have defined them in detail:

Query:

```
type Query {
	// handle query for doctor related information
	doctor(name: String, id: ID): Doctor

	// handle query for appointment related information
	appointment(doctorId: ID!, appointmentId: ID!): Appointment

	// handle query for calendar information for a given date for a doctor
	calendar(date: String!, doctorId: ID!): Calendar
}
```

Mutation:

```
type Mutation {
	// mutation to book an appointment
	bookAppointment(doctorId: ID!, patientName: String!, date: String!,
		timeslot: ID!): Appointment

	// mutation to cancel an appointment
	cancelAppointment(appointmentId: ID!, doctorId: ID!) Appointment

	// mutation to update an appointment
	updateAppointment(appointmentId: ID!, patientName: String):
			Appointment
}
```

Based on above query and mutations, we define the expected capabilities:

1. Get details for a doctor (name, clinic, specialty)
   To implement this, we use the *doctor* query. Doctor information can be searched based on the name or the id of the doctor.

```
query GetDoctorDetail {
	doctor(name: String, id: ID) {
		id
		name
		clinic
		specialty
	}
}
```

   Sample response as viewed on the client-side:

```
{
	“data”: {
		“doctor”: {
			“id”: 1001,
			“name” : “Doctor.X”,
			“clinic”: “New Clinic”,
			“specialty”: “Dermatologist”
		}
	}
}
```

2. Get doctor's available timeslots for today
   To implement this capability, we create a query where available timeslots can be found for a doctor on any given date. The query will take two mandatory inputs, date, and doctor id to find the calendar information for a given date.

```
query GetAvailableTimeslotForDoctorForADate {
	calendar(date: String!, doctorId: ID!) {
		date
		doctor {
			name
			clinic
			specialty
		}
		availableTimeslots(date: String!, doctorId: ID!) {
			startTime
			endTime
		}

	}
}
```

   Sample response as viewed on the client-side:

```
{
	"data": {
		"calendar" : {
			"date": "12-12-2022",
			"doctor": {
				"name": "Doctor.X",
				"clinic": "New Clinic"
				"specialty": "Dermatologist
			}
			"availableTimeslots": [
				{
					"startTime": "10:30AM",
					"endTime": "11:00AM"

				},
				{
					"startTime": "11:00AM",
					"endTime": "11:30AM"

				}
			]
		}
	}
}
```

3. Book an appointment with doctor for today.
   This functionality can be implemented by using the *bookAppointment* query where we pass all the information regarding an appointment such as doctor id, patient name, date, and time slot to create a calendar event.

```
mutation BookAppointment {
	bookAppointment (doctorId: ID!, patientName: String!, date:
		String!, timeslot: ID!) {
		id
		doctor {
			name
			clinic
			specialty
		}
		patientName
		timeslot {
			startTime
			endTime
		}
		date
	}
}
```

Sample response as viewed on the client-side:

```
{
	“data”: {
		“bookAppointment” : {
			“id”: 2001,
			“date”: “12-12-2022”,
			“doctor”: {
				“name”: “Doctor.X”,
				“clinic”: “New Clinic”,
				“specialty”: “Dermatologist
			},
			patientName: “Mr.A”,
			timeslot: {
				startTime: “12:00PM”,
				endTime: “12:30PM”
			},
			“cancelled”: false
		}
	}
}
```

4. Cancel an appointment.
   To achieve this feature, we use *cancelAppointment* query which returns the schema type *Appointment.* It needs two mandatory fields, appointment id and doctor id, to successfully delete an event from the doctor's calendar.

```
mutation CancelAppointment {
	cancelAppointment (appointmentId: ID!, doctorId: ID!) {
		doctor {
			name
			clinic
			specialty
		}
		patientName
		timeslot {
			startTime
			endTime
		}
		date
	}
}
```

   Sample response as viewed on the client-side:

```
{
	"data": {
		"cancelAppointment" : {
			"date": "12-12-2022",
			"doctor": {
				"name": "Doctor.X",
				"clinic": "New Clinic",
				"specialty": "Dermatologist
			},
			patientName: "Mr.A",
			timeslot: {
				startTime: "12:00PM",
				endTime: "12:30PM"
			},
			"cancelled": true
		}
	}
}
```

5. Update name of the patient in an appointment:
   Above capability comes under the bigger umbrella of updating an appointment therefore we implement an *UpdateAppointment mutation,* which takes mandatory appointment id, doctor id along with other optional fields which need to be updated and updates the event accordingly in the doctor's calendar.

```
mutation UpdateAppointment {
	updateAppointment (appointmentId: ID!, doctorId: ID!, patientName:
			String, timeslotId: ID, date: String) {
		id
		doctor {
			name
			clinic
			specialty
		}
		patientName
		timeslot {
			startTime
			endTime
		}
		date
	}
}
```

   Sample response as viewed on the client-side:

```
{
	"data": {
		"updateAppointment" : {
			"id": 2001,
			"date": "12-12-2022",
			"doctor": {
				"name": "Doctor.X",
				"clinic": "New Clinic",
				"specialty": "Dermatologist
			},
			patientName: "Mr.A",
			timeslot: {
				startTime: "12:00PM",
				endTime: "12:30PM"
			},
			"cancelled": false
		}
	}
}
```

Task 1.2: Testcases design

Following testcases can be designed to test the above API implementation:

Testcase Identifier	Testcase Description	Inputs	Expected Output	Remarks

BookAppointmentSuccess	This is a testcase to test if correct and required inputs are passed, appointment is booked correctly.	Valid inputs of the format shown below are accepted.
		
		{
		doctorId: int,
		patientName: String,
		date: String,
		timeslot: int
		}
			Expected output would be of the type of mutation which includes information about booked appointment.
			
			{
			“data”: {
			“bookAppointment”:{
			“id”: 2001,
			“date”: “12-12- 2022”,
			“doctor”: {
			“name”: “Doctor.X”,
			“clinic”: “New Clinic”,
			“specialty”: “Dermatologist
			},
			“patientName”: “Mr.A”,
			“timeslot”: {
			startTime: “12:00PM”,
			endTime: “12:30PM”
			}
			}
			}
			}
				The output matches the appointment schema type. Testcase identifier defines the names that will be used while writing tests.

BookAppointmentFail	This is a testcase to test if incomplete inputs are passed, appointment is not booked, and user is notified of the error.	Invalid inputs with one or more fields missing
		
		{
		patientName: String,
		date: String,
		timeslot: int
		}
			Expected output would include an error message:
			
			{
			“data”: {
			“error”:{
			“message”: “Doctor id is missing”
			}
			}
			}
				Error handling needs to be done in the GraphQL API on server side.

UpdateAppointmentSuccess	This is a testcase to test if correct and required inputs are passed, appointment is updated correctly.	Valid inputs include mandatory fields doctor id and appointment id. Rest all fields are optional.
		
		{
		“doctorId”: 1001,
		“patientName”:“Mr.Y”,
		“appointmentId: 2001
		}
			Expected output would be of the type of mutation which includes information about the updated appointment.
			
			{
			“data”: {
			“updateAppointment”:{
			“id”: 2001,
			“date”: “12-12- 2022”,
			“doctor”: {
			“name”: “Doctor.X”,
			“clinic”: “New Clinic”,
			“specialty”: “Dermatologist
			},
			“patientName”: “Mr.Y”,
			“timeslot”: {
			startTime: “12:00PM”,
			endTime: “12:30PM”
			}
			}
			}
			}
				Update appointment can also be used to cancel an event but separate endpoint has been made for it to remove ambiguity and mistakes.

UpdateAppointmentFail	This is a testcase to test if incomplete inputs are passed, appointment is not updated, and user is notified of the error.	Invalid inputs include one or more missing mandatory fields, i.e., doctor id and appointment id.
		
		{
		“doctorId”: 1001,
		“patientName”:“Mr.Y”
		}
			Expected output would include an error message:
			
			{
			“data”: {
			“error”:{
			“message”: “Appointment id is missing”
			}
			}
			}
				For error, response codes can be enumerated on the server side to make message more understandable. For instance, if wrong appointment id was passed, it can

				say NOT_FOUND to indicate resource is not available.

CancelAppointmentSuccess	This is a testcase to test if an appointment is deleted for a given appointment id and doctor id.	Valid inputs include an appointment id and a doctor id.
		
		{
		“doctorId”: 1001,
		“appointmentId: 2001
		}
			Expected output would include details about the cancelled appointment. The output will have cancelled key set as true.
			
			{
			“data”: {
			“cancelAppointment”:{
			“date”: “12-12- 2022”,
			“doctor”: {
			“name”: “Doctor.X”,
			“clinic”: “New Clinic”,
			“specialty”: “Dermatologist
			},
			“patientName”: “Mr.A”,
			“timeslot”: {
			startTime: “12:00PM”,
			endTime: “12:30PM”
			}
			“cancelled”: true
			}
			}
			}
				cancelled key will become true if appointment is cancelled successfully.

CancelAppointmentFail	This is a testcase to test if incomplete inputs are passed, appointment is not cancelled, and user is notified of the error.	Invalid inputs include one or more missing mandatory fields, i.e., doctor id and appointment id.
		
		{
		“doctorId”: 1001,
		“patientName”:“Mr.Y”
			Expected output would include an error message:
			
			{
			“data”: {
			“error”:{
			“message”: “Appointment id is missing”
			}
			}
				Error message should handle all types of errors.

		}	}	

DoctorQuerySuccess	Test to check if all doctor data is successfully retrieved or not. Doctor detail can be fetched based on the name or the id passed in the argument.	Valid inputs include at least one of doctor name or a doctor id.
		
		{
		“doctorId”: 1001,
		}
			Expected output will include all the doctor information. Example output:
			
			{
			“data”: {
			“doctor”: {
			“id”: 1001,
			“name” : “Doctor.X”,
			“clinic”: “New Clinic”,
			“specialty”: “Dermatologist”
			}
			}
			}	

DoctorQueryFail	Test to check if API throws error if invalid query is made.	Invalid inputs include incorrect key.
		
		{
		“invalidKey”: 1001,
		}
			The GraphQL API will throw an error that the given query is invalid.
			
			{
			“data”: {
			“error”:{
			“message”: “Invalid query”
			}
			}
			}	

AppointmentQuerySuccess	Test to check if appointment query fetches appointment details based on the valid input.	Valid inputs include both doctorId and appointmentId.
		
		{
		“doctorId”: 1001,
		“appointmentId”: 2001
		}
			Expected output will include all the appointment information. Example output:
			
			{
			“data”: {
			“appointment”:{
			“id”: 2001,
			“date”: “12-12- 2022”,
			“doctor”: {
			“name”: “Doctor.X”,	

			“clinic”: “New Clinic”,
			“specialty”: “Dermatologist
			},
			“patientName”: “Mr.Y”,
			“timeslot”: {
			startTime: “12:00PM”,
			endTime: “12:30PM”
			}
			}
			}
			}	

AppointmentQueryFail	Test to check if all appointment query fails if we miss a required argument in the test.	Invalid inputs include missing appointment id or doctor id.
		
		{
		“appointmentId”: 2001,
		}
			The GraphQL API will throw an error that the given query is missing doctor id.
			
			{
			“data”: {
			“error”:{
			“message”: “Doctor id is missing”
			}
			}
			}
				Tests should also be written to handle the validity of inputs, and when resources do not exist.

CalendarQuerySuccess	Test to check for a given date and doctor id all the relevant information is fetched for the schema type calendar.	Valid input includes date for which we need to check the calendar for and the doctor id for which we need to fetch the detail.
		
		{
		“doctorId”: 1001,
		“date”: “12-12-2022”
		}
			The GraphQL API will return expected output as shown below with all the details about a given date for a doctor.
			
			{
			“data”: {
			“calendar” : {
			“date”: “12-12-2022”,
			“doctor”: {
			“name”: “Doctor.X”,
			“clinic”: “New Clinic”
			“specialty”: “Dermatologist
			}
			“availableTimeslots”: [
			{	

			“startTime”: “10:30AM”,
			“endTime”: “11:00AM”
			},
			{
			“startTime”: “11:00AM”,
			“endTime”: “11:30AM”
			}
			]
			}
			}
			}	

CalendarQueryFail	Test to check if query fails if the date is missing in the query for bringing calendar data for a given date.	Invalid inputs include missing appointment id or doctor id.
		
		{
		“doctorId”: 1001,
		}
			The GraphQL API will throw an error that the given query is missing date.
			
			{
			“data”: {
			“error”:{
			“message”: “Date is missing”
			}
			}
			}	