Appointment Booking Challenge

Overview

We want to build an appointment booking system that allows customers to schedule appointments with our sales managers to discuss one or more of our products. For the MVP, we will have a website that displays available appointment slots that a customer can choose from.

The goal of this project is to implement the backend for this system. We need an endpoint that returns the available appointment slots for a customer.

There are a few rules we need to consider when checking for available appointment slots for a customer:

- Each slot corresponds to a one-hour appointment
- Slots can have overlapping time ranges. For example, it is possible to have the following three slots:
  - 10:30 - 11:30
  - 11:00 - 12:00
  - 11:30 - 12:30
- A sales manager CANNOT be booked for two overlapping slots at the same time. For example, if a sales manger has a slot booked at 10:30 - 11:30, then the 11:00 - 12:00 cannot be booked anymore.
- Customers are matched to sales managers based on specific criteria. A slot CANNOT be booked by a customer if the sales manager does not match any of these three criteria:
  - Language. Currently we have 2 possible languages: German, English
  - Product(s) to discuss. Currently we have 2 possible products: SolarPanels, Heatpumps
  - Internal customer rating. Currently we have 3 possible ratings: Gold, Silver, Bronze.
- Customers can book one appointment to discuss multiple products

Requirements

Design and implement a REST endpoint in any language of your choice that:

- Listens for POST requests on this route: `http://localhost:3000/calendar/query`
- Connects to the provided Postgres database instance
- Receives a request body in this format:

```
{
  "date": "2024-05-03",
  "products": ["SolarPanels", "Heatpumps"],
  "language": "German",
  "rating": "Gold"
}
```

- Returns a response with an array of available slots that can be booked by the customer in this format

```
[
  {
      "available_count": 1,
      "start_date": "2024-05-03T10:30:00.00Z"
  },
  {
      "available_count": 2,
      "start_date": "2024-05-03T12:00:00.00Z"
  }
]
```

NOTES:

- You can use any language, framework, library of your choice for this challenge.
- The system should not book appointments in this challenge; your focus is returning available slots.

- We provide you with a docker database already populated with data. You are not allowed to modify the database structure of the database in any way but you can create indexes or views if you think it is necessary. The docker database can be downloaded here: https://enpalcorepgtechiv.blob.core.windows.net/tech-interview/backend/20241028_34f4060c/Take_Home_Challenge_Resources.zip

Database Schema

The provided database has the following schema:

Table: sales_managers

Column Name	Column Type	Comment
id (PK)	serial	ID of the sales manager
name	varchar(250)	Full name of sales manager
languages	array(varchar(100))	List of languages spoken by sales manager
products	array(varchar(100))	List of products the sales manager can work with
customer_ratings	array(varchar(100))	List of customer ratings the sales manager can work with

Table: slots

Column Name	Column Type	Comment
id (PK)	serial	ID of the slot
start_date	timestampz	Start date and time of the slot
end_date	timestampz	End date and time of the slot
booked	bool	Value indicating whether the slot has already been booked
sales_manager_id (FK)	integer	ID of the sales manager the slot belongs to

Getting Started

Download the resources for this challenge here https://enpalcorepgtechiv.blob.core.windows.net/tech-interview/backend/20241028_34f4060c/Take_Home_Challenge_Resources.zip. This is a zip file that contains two folders:

- `database`: This folder contains a `Dockerfile` that can be used to start a Postgres database server and an `init.sql` file that initializes the database and preloads it with data.
- `test-app`: This folder contains a node application that can be used to run test scenarios to verify your application.

Setup the database

Extract the `Take_Home_Challenge_Resources.zip` file and run the following commands in the `database` folder. This requires that you have docker installed on your local environment

```
docker build -t enpal-coding-challenge-db .
docker run --name enpal-coding-challenge-db -p 5432:5432 -d enpal-coding-challenge-db
```

Once the docker container is up and running, ensure you can connect to it using your favourite DB query tool (e.g.: DBeaver or pgAdmin). The default connection string is `postgres://postgres:mypassword123!@localhost:5432/coding-challenge`

If you want to use a local database installation instead, you also can get the `init.sql` file and run it in your local database.

Setup tests

Extract the `Take_Home_Challenge_Resources.zip` file and run the following commands in the `test-app` folder. This requires that you have node installed on your local environment

```
npm install
npm run test
```

The tests try to connect to an endpoint running on http://localhost:3000/calendar/query and run several test scenarios. Since that is not probably running yet the tests will fail.

You can inspect the `test.js` file and see some example requests and the expected responses.

Start coding

You can now create an api in your language of choice that fulfils the requirements.

How to submit the solution

The solution should contain:

- your application code
- docker setup that starts the database
- instructions on how to run your application.

You can send this to us as a zip file or push this to a github repository and send us the link.

Your solution must connect to the database in the docker container and then we'll run the same tests provided to you. We might ALSO run additional tests, such as loading thousands of records in the database to assert the application is performant enough.

Evaluation Criteria

Please note that we will be evaluating your solution not just based on correctness but also on the following criteria. We place a high importance on these criteria, and we strongly encourage you to carefully consider them as you develop your solution:

- Accuracy in adhering to the specified rules.
- Efficiency and performance of the api endpoint.
- Clarity, readability and testability of the code.
- Handling of edge cases and error conditions.