Coding Challenge example

A company sells various products and wants to facilitate appointment bookings for customers.
Customers can book appointments through a website that displays available slots.
Each slot corresponds to a one-hour appointment.
There are specific rules for displaying these slots.
The system matches customers with appropriate sales managers based on certain traits.

Getting started

You can download the resources from here: https://enpalcorepgtechiv.blob.core.windows.net/tech-interview/2024_06%20Take%20home%20challenge%20resources.zip

The zip file contains the following structure:

- db
  - Dockerfile
  - init.sql
- tests
  - package.json
  - test.js

Set up the database

The first thing to do is to start the database. For that you need docker installed in your local computer.

```
cd ./database/
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://postgress:mypassword123!@localhost:5432/coding-challenge`

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

Set up tests

To test the implemented solution, a test application is provided. To run this application, node is required.

```
npm install
npm run test
```

The tests try to connect to an endpoint running on `http://localhost:3000/calendar/query` and run several 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.

Requirements

Your task is to design and implement a rest endpoint in your language of choice that:

- listens for post requests on this route: `http://localhost:3000/calendar/query`
- connects to the provided database

- receives the request body described below
- returns the response content described below

Url

```
[POST] http://localhost:3000/calendar/query
```

Input

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

Output

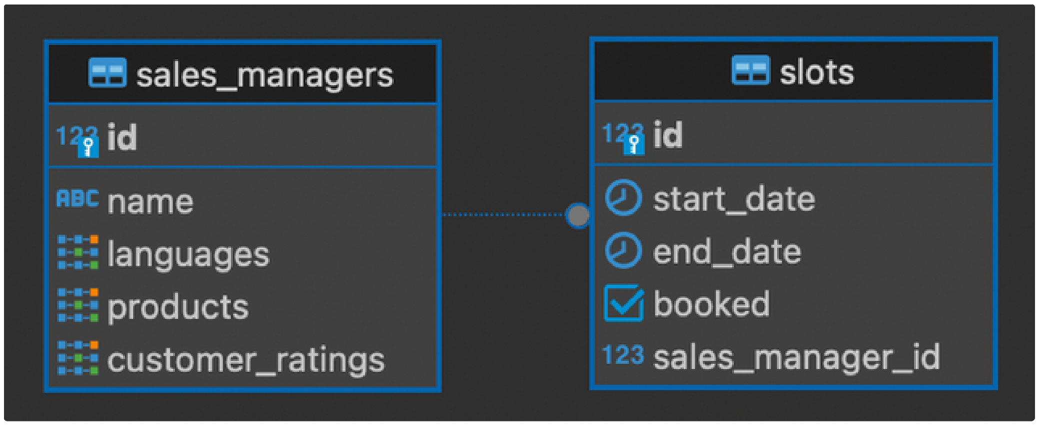
```
[
  {
    "available_count": 1,
    "start_date": "2024-05-03T10:30:00.00Z"
  },
  {
    "available_count": 2,
    "start_date": "2024-05-03T12:00:00.00Z"
  }
]
```

Rules

- Only return data if there is at least one sales manager that matches the language, products and rating of the input parameter.
- Display the count of how many sales managers are available per slot.
- A slot can be booked or not. Available slots that conflict with other booked slots should not be counted.
- A customer is matched to a sales manager when the language, rating and products of input parameter intersect with the ones a sales manager has.

Database

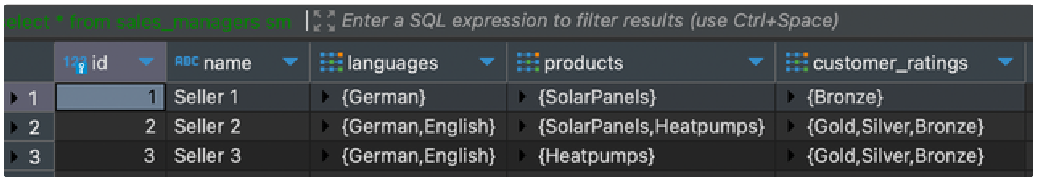
The provided database is already populated with data. You can look into the `init.sql` file in the docker project to see initialisation scripts. You cannot modify the database structure in any way but you can create indexes or views if you think it is necessary.



Sales Managers table

- **id** (serial) - id of the sales manager
- **name** (varchar(250)) - full name of the sales
- **languages** (array(varchar(100))) - list of languages spoken by the Sales Manager
  - German
  - English
- **products** (array(varchar(100))) - list of products the sales manager can work with
  - SolarPanels
  - Heatpumps
- **customer_ratings** (array(varchar(100))) - list of customer ratings the sales manager can work with
  - Gold
  - Silver
  - Bronze

There are three Sales Managers:



```
select * from sales_managers sm
```

Enter a SQL expression to filter results (use Ctrl+Space)

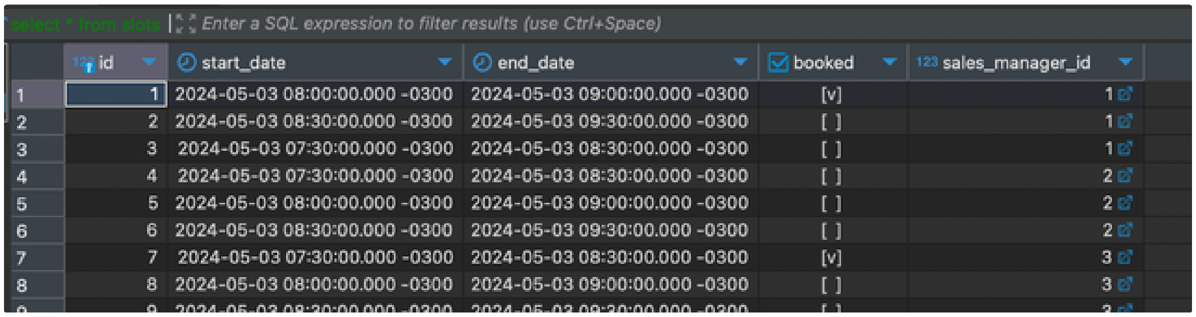
	id	name	languages	products	customer_ratings
1	1	Seller 1	{German}	{SolarPanels}	{Bronze}
2	2	Seller 2	{German,English}	{SolarPanels,Heatpumps}	{Gold,Silver,Bronze}
3	3	Seller 3	{German,English}	{Heatpumps}	{Gold,Silver,Bronze}

- **Seller 1** only speaks German, can sell SolarPanels only and, since he is a junior salesman, can only work with Bronze rated customers
- **Seller 2** speaks both German and English, can sell Solar Panels and Heatpumps and can work with any customer rating.
- **Seller 3** can also speak both languages and work with all customer ratings, but only sells Heatpumps.

Slots table

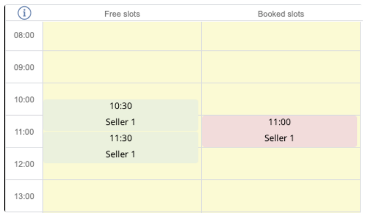
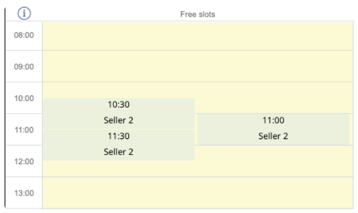
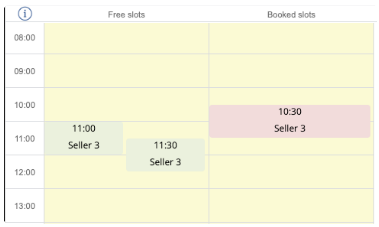
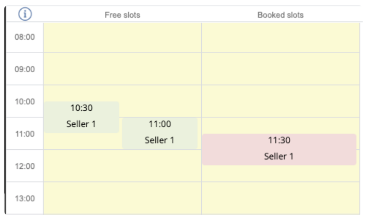
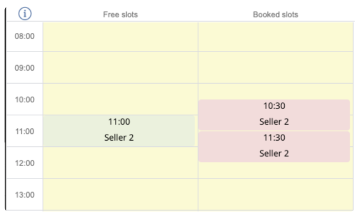
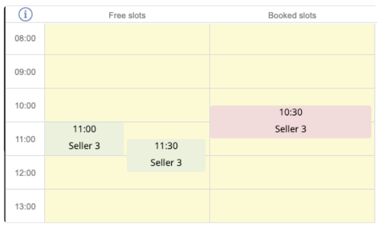
- **id** (serial) - id of the record
- **start_date** / **end_date** (timestamptz) - dates indicating the start and end times of the slot
- **booked** (bool) - a value indicating whether the slot is booked to a customer or not.
- **sales_manager_id** (int) - a value indicating to which sales manager the slot belongs to.

The slots table is populated with 3 slots on 2 days for each sales manager:



```
select * from slots | Enter a SQL expression to filter results (use Ctrl+Space)
   id  start_date                     end_date                       booked  sales_manager_id
1   1  2024-05-03 08:00:00.000 -0300  2024-05-03 09:00:00.000 -0300  [v]     1
2   2  2024-05-03 08:30:00.000 -0300  2024-05-03 09:30:00.000 -0300  [ ]     1
3   3  2024-05-03 07:30:00.000 -0300  2024-05-03 08:30:00.000 -0300  [ ]     1
4   4  2024-05-03 07:30:00.000 -0300  2024-05-03 08:30:00.000 -0300  [ ]     2
5   5  2024-05-03 08:00:00.000 -0300  2024-05-03 09:00:00.000 -0300  [ ]     2
6   6  2024-05-03 08:30:00.000 -0300  2024-05-03 09:30:00.000 -0300  [ ]     2
7   7  2024-05-03 07:30:00.000 -0300  2024-05-03 08:30:00.000 -0300  [v]     3
8   8  2024-05-03 08:00:00.000 -0300  2024-05-03 09:00:00.000 -0300  [ ]     3
9   9  2024-05-03 08:30:00.000 -0300  2024-05-03 09:30:00.000 -0300  [ ]     3
```

For the sake of simplicity we will work with two days only and with the following calendar distributions.
The green slots are free and the red ones are booked.
Pay attention to the conflicts between booked and free slots. For example, for the Seller 1 on Monday the Booked slot intersects with the two free slots. This means the Seller 1 is not available on Monday.

	Seller 1 language: German products: [SolarPanels] customer ratings: [Bronze]	**Seller 2** language: German, English products: [SolarPanels, Heatpumps] customer ratings: [Gold, Silver, Bronze]	**Seller 3** language: German, English products: [Heatpumps] customer ratings: [Gold, Silver, Bronze]
Monday 03.05.2024	 Free slots Booked slots 08:00 09:00 10:00 11:00 12:00 13:00 10:30 Seller 1 11:30 Seller 1 11:00 Seller 1 Fully booked, since the booked slot intersects with the free slots	 Free slots 08:00 09:00 10:00 11:00 12:00 13:00 10:30 Seller 2 11:30 Seller 2 11:00 Seller 2 All slots are bookable	 Free slots Booked slots 08:00 09:00 10:00 11:00 12:00 13:00 11:00 Seller 3 11:30 Seller 3 10:30 Seller 3 Only the 11:30 slot is bookable since the 10:30 slot intersects with the 11:00
Tuesday 04.05.2024	 Free slots Booked slots 08:00 09:00 10:00 11:00 12:00 13:00 10:30 Seller 1 11:00 Seller 1 11:30 Seller 1 Only the 10:30 slot is bookable since the booked slot at 11:30 intersects with the free slot at 11:00	 Free slots Booked slots 08:00 09:00 10:00 11:00 12:00 13:00 11:00 Seller 2 10:30 Seller 2 11:30 Seller 2 Fully booked since the free slot intersect with the other booked slots	 Free slots Booked slots 08:00 09:00 10:00 11:00 12:00 13:00 11:00 Seller 3 11:30 Seller 3 10:30 Seller 3 Only the 11:30 slot is bookable since the booked slot at 10:30 intersects with the free slot at 11:00

How to submit the solution

Send a zip file with your source code and a docker project that starts the database and api on `http://localhost:3000/calendar/query`

If additional steps are needed send them in a README.md file.

We will run the provided test application, evaluate the code and execute additional tests for more advanced scenarios.

Notes

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

Evaluation Criteria:

Your solution will be evaluated based on:

- 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.