Pollinate – Technical Questions

Section 1

The Scope of this project is to call an API which will push the data to the database. The API is written in Java and built in Spring boot.

Table used to store the data is built in MS SQL.

Components	Language/ Database Type
Database	MSSQL
API	Java/Spring boot

Detailed Design

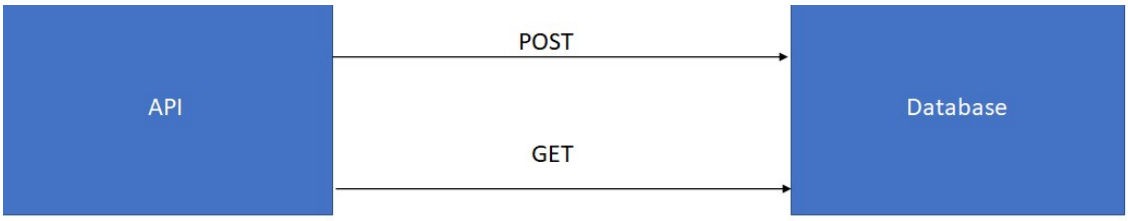
The API built as part of the project inserts the timestamp into the Product table which is stored in the MS SQL database.

The product table is replicated in the API as a model class where the column names are defined as the variables in the class.

The API has two options to the user:

- GET the data from the Product table by doing a get call to the database.
- POST the data to the product table by doing a post call and passing 2 Parameters in the request which are:
    - ID: ID of the product which needs to be inserted
    - Timestamp: Timestamp which is passed to the API

Below is the diagrammatical representation of the same:



Classes\Interface in API:

Classes	Type	Description

Product	Model	A Java view of the class for the table which exists in Database
UserRepository	Java Interface	An interface which extends JPA's CRUD repository to do database operations
MainController	Java Controller class	A Java class to help build the URI's for the project which the user is going to hit in the application
PollinateApplication	Java Main Class	A main class to run the Spring boot Application

Created the below table Product with table structure as mentioned in the MSSQL data:

```
33
34 •   describe sakila.product
35
```

Result Grid | Filter Rows: | Export: | Wr

	Field	Type	Null	Key	Default	Extra
▶	ID	int	NO	PRI	NULL	
	PRODUCT_TIME	datetime	YES		NULL	

URI

Item	URI	Parameters
API -> Database	GET http://localhost:8080/demo/all	
API -> Database	POST http://localhost:8080/demo/app	ID
		Timestamp

How to Run the API

The API is built to be executable on your device if the following pre-requisites are met.

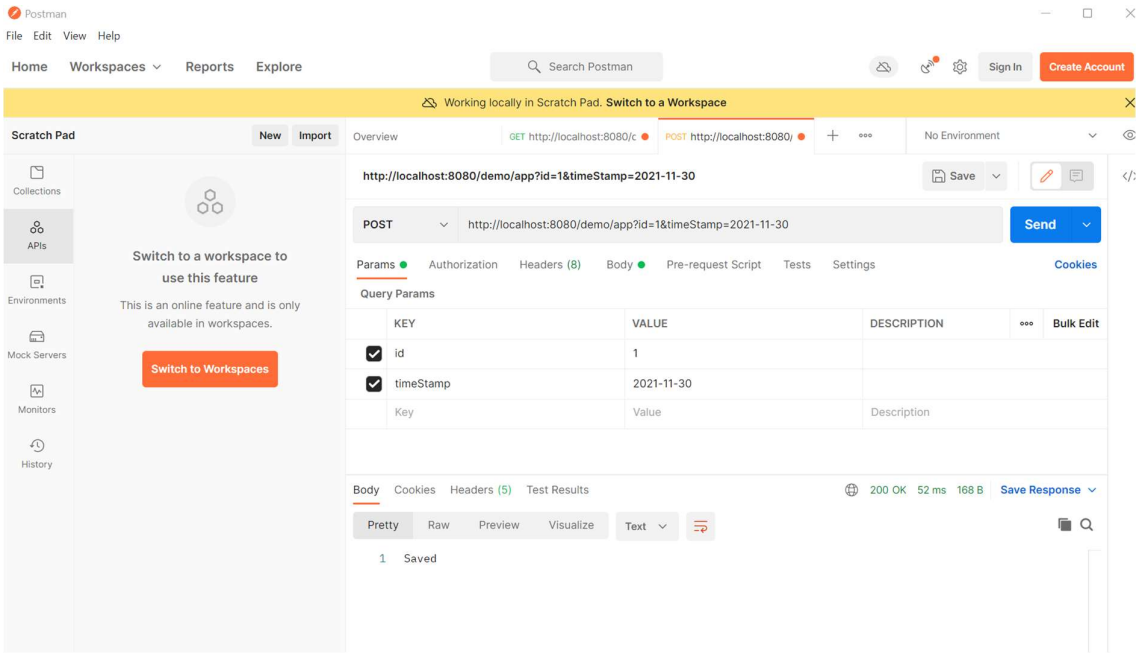
- Nothing running on the port 8080 as this is the port where the API is being executed.
- A database built in your local environment whose details are included in the application.properties file of the code

To run the API a user can either use a Postman or a curl command

Postman

To execute via postman the user needs to make one of the calls mentioned above in the URI section of the design.

To post the data to the database the user needs to mention two columns in the param section of the API. An example for the same is attached below:



Curl

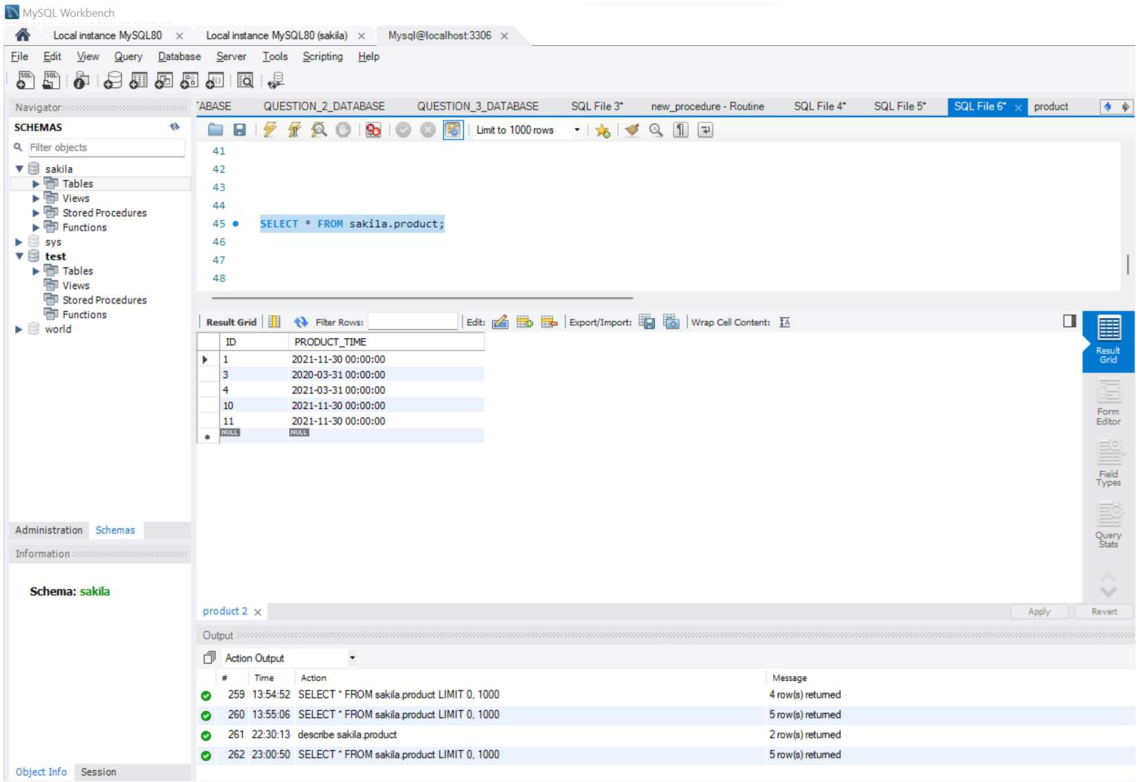
To use the curl command the user can execute the below command:

curl -X POST http://localhost:8080/demo/app?id=2&timeStamp=2021-11-30

```
C:\Users\kanav>curl --data "id=1&timeStamp=2021-11-30" http://localhost:8080/demo/app
Saved
C:\Users\kanav>curl --data "id=10&timeStamp=2021-11-30" http://localhost:8080/demo/app
Saved
C:\Users\kanav>curl --data "id=11&timeStamp=2021-11-30" http://localhost:8080/demo/app
Saved
C:\Users\kanav>curl --data "id=11&timeStamp=2021-11-30" http://localhost:8080/demo/app
```

MSSQL

Below screenshot shows the data in the database



Proof of concept along with ReadMe document capturing all the relevant steps to be followed for execution in uploaded successfully in the below Github repository

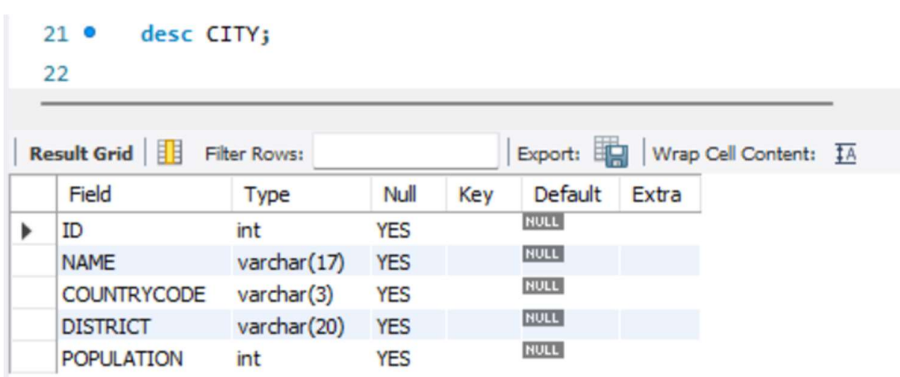
Github repository:

https://github.com/chethuaries19/pollinate.git

Section 2

1. Query all columns for a city in CITY with the *ID* 1661.

Created table with name City and Attributes, datatypes as mentioned



Field	Type	Null	Key	Default	Extra
ID	int	YES		NULL	
NAME	varchar(17)	YES		NULL	
COUNTRYCODE	varchar(3)	YES		NULL	
DISTRICT	varchar(20)	YES		NULL	
POPULATION	int	YES		NULL	

Inserted below values into the table

```
23 •    SELECT * FROM CITY;
24
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ID	NAME	COUNTRYCODE	DISTRICT	POPULATION
1661	AUSTRALIA	061	MELBOURNE	2037832
1662	INDIA	091	BANGALORE	98293833

Query to fetch all the columns of city with ID=1661

```
18      /* TO FETCH ALL THE DETAILS OF CITY TABLE WITH ID=1661*/
19 •    SELECT * FROM CITY WHERE ID=1661;
20
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ID	NAME	COUNTRYCODE	DISTRICT	POPULATION
1661	AUSTRALIA	061	MELBOURNE	2037832

2. Write a query that prints a list of employee names (i.e.: the *name* attribute) for employees in Employee having a salary greater than per month who have been employees for less than months. Sort your result by ascending *employee_id*

Created table with name EMPLOYEE and Attributes, datatypes as mentioned

```
9 •    desc EMPLOYEE;
```

Result Grid | Filter Rows: | Export: | Wrap Cel

Field	Type	Null	Key	Default	Extra
EMPLOYEE_ID	int	YES		NULL	
NAME	varchar(255)	YES		NULL	
MONTHS	int	YES		NULL	
SALARY	int	YES		NULL	

Inserted below values into the table

```
select * from EMPLOYEE;
```

Result Grid | Filter Rows: | Export:

EMPLOYEE_ID	NAME	MONTHS	SALARY
12228	ROSE	15	1968
33645	ANGELA	1	3443
45692	FRANK	17	1608
56118	PATRICK	7	1345
59725	LISA	11	2330
74197	KIMBERLY	16	4372
78454	BONNIE	8	1771
83565	MICHAEL	6	2017
98607	TODD	5	3396
99989	JOE	9	3573

To print a list of employee names (i.e.: the *name* attribute) for employees in Employee having a salary greater than per month who have been employees for less than months

/TO Print EMPLOYEE names WITH SALARY >some value AND MONTHS< some value- have considered salary>2000 and Months<10 as per the output mentioned in the question sheet

```
SELECT
    NAME
FROM
    EMPLOYEE
WHERE
    SALARY > 2000 AND MONTHS < 10
ORDER BY EMPLOYEE_ID ASC;
```

Result Grid | Filter Rows: | Export:

NAME
ANGELA
MICHAEL
TODD
JOE

3.1. Query an *alphabetically ordered* list of all names in OCCUPATIONS, immediately followed by the first letter of each profession as a parenthetical (i.e.: enclosed in parentheses). For example: AnActorName(A), ADoctorName(D), AProfessorName(P), and ASingerName(S).

Created table with name OCCUPATIONS and Attributes, datatypes as mentioned

```
7 •  desc OCCUPATIONS;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Co

Field	Type	Null	Key	Default	Extra
NAME	varchar(255)	YES		NULL	
OCCUPATION	varchar(255)	YES		NULL	

Inserted below values into the table

```
8 •  select * from OCCUPATIONS;
```

Result Grid | Filter Rows:

NAME	OCCUPATION
Samantha	Doctor
Julia	Actor
Maria	Actor
Meera	Singer
Ashley	Professor
Ketty	Professor
Christeen	Professor
Jane	Actor
Jenny	Doctor
Priya	Singer

/*Query to order the name alphabetically in specified format*/

```
24    /*Query to order the occurences alphabetically in specified format*/
25 •  SELECT
26        CONCAT(NAME, '(', LEFT(OCCUPATION, 1), ')') AS NameOcc
27    FROM
28        OCCUPATIONS
29    ORDER BY Name;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

NameOcc
Ashley(P)
Christeen(P)
Jane(A)
Jenny(D)
Julia(A)
Ketty(P)
Maria(A)
Meera(S)
Priya(S)
Samantha(D)

3.2. Query the number of ocurrences of each occupation in OCCUPATIONS. Sort the occurrences in *ascending order*, and output them in the following format:

```
There are a total of [occupation_count] [occupation]s.
```

where [occupation_count] is the number of occurrences of an occupation in OCCUPATIONS and [occupation] is the *lowercase* occupation name. If more than one *Occupation* has the same [occupation_count], they should be ordered alphabetically.

```sql
SELECT
    CASE
        WHEN
            COUNT(OCCUPATION) > 1
        THEN
            CONCAT('There are total of ', COUNT(OCCUPATION), ' ', LOWER(OCCUPATION), 's')
        ELSE CONCAT('There are total of ', COUNT(OCCUPATION), ' ', LOWER(OCCUPATION))
    END AS NameOcc
FROM
    OCCUPATIONS B
GROUP BY OCCUPATION
ORDER BY COUNT(OCCUPATION) , Occupation ASC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

NameOcc
There are total of 2 doctors
There are total of 2 singers
There are total of 3 actors
There are total of 3 professors

4. *P(R)* represents a pattern drawn by Julia in *R* rows. The following pattern represents *P(5)*:

```
*
* *
* * *
* * * *
* * * * *
```

Write a query to print the pattern *P(20)*.

Procedure to create the above pattern

```
DROP PROCEDURE IF EXISTS starprint;
DELIMITER $$
CREATE PROCEDURE starprint()
BEGIN
    DECLARE X INT(10) DEFAULT 1;
```

Result Grid | Filter Rows: | Export: | Wrap

pattern
*
**
***
****
*****
******
****
****
****
****
****
****
****
****
****
****
****
****
****
****

Or by using information_schema.Tables

```
/*Simple query using information_schema.tables*/
set @number = 0;
select repeat('* ', @number := @number + 1) from information_schema.tables limit 20;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

repeat('*', @number := @number + 1)
*
**
***
****
*****
******
****
****
****
****
****
****
****
****
****
****
****
****
****
****

5. *P(R)* represents a pattern drawn by Julia in *R* rows. The following pattern represents *P(5)*:

```
* * * * *
* * * *
* * *
* *
*
```

Write a query to print the pattern *P(20)*.

Procedure to create the above pattern

```
5  BEGIN
6      DECLARE X INT(10) DEFAULT 20;
7      DECLARE P VARCHAR(255) DEFAULT " *";
8      CREATE TEMPORARY TABLE TEMP (`rowno` INT, `pattern` VARCHAR(255));
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

pattern
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****

Or by using information_schema.Tables

```
22    /*Simple query using information_schema.tables*/
23 •  set @number = 21;
24 •  select repeat('* ', @number := @number - 1) from information_schema.tables limit 20;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

repeat('*', @number := @number - 1)
****
****
****
****
****
****
****
****
****
****
****
****
****
****
******
*****
****
***
**
*