

Technical Report on

# Normalisation And Connecting to A Database Using A Web App

Cinthia, Wei  
2023/11/14

## Introduction

The report aims to provide answers to three questions: what makes a relational database “relational”? How to develop a web application? And why do we need web applications? Therefore, the content surrounds three key parts: conceptualize relationships between tables and data in a relational database, develop a database-backed web application using Flask and other software packages, and exemplify the potential uses of web application in business context with a case study. The report also contains step-by-step instructions and troubleshooting. Windows system is used in all examples.

## Question 1

a)

Functional dependency is a constraint that specifies the relationship of one attribute to another attribute in a relational table. It describes the one-way relationship that values of one attribute determine the values of another attribute, and reverse case is not guaranteed. Functional dependency is an important concept in normalization and helps to determine suitable keys to search for data or make queries.

Suppose there is a table with two columns, X and Y, values in X are denoted by x and values in Y are denoted by y. For instance, x1 represents the value in first row in column X; y2 represents the value in second row in column Y, so on so forth.

The goal is to check the if Y is functionally dependent on X, meaning that, each unique x corresponds to one y. There can be repetitive values within column X and within column Y, but identical x values must all have the same y value. Y is not necessarily a function output of X or computed based on X.

X	Y
1	60
2	55
3	80
4	75

Table 1

In table 1, x1 = 1 corresponds to y1 = 60, x2 = 2 corresponds to y2 = 55, x3 = 3 corresponds to y3 = 80, x4 = 4 corresponds to y4 = 75. It is found that each x associates with a different y, so x can be used to search for y.

X	Y
1	60
2	55
3	80
4	75
2	55

Table 2

If a new row of data is added as shown in table 2, functional dependency of Y on X still exists, since even though there are two identical x values (x2 and x5 both equals 2), they all correspond to the same y value, 55.

X	Y
1	60
2	55
3	80
4	75
5	80

Table 3

In table 3, functional dependency of Y on X still exists despite repetition in y values. We are able to tell y value based on any x.

X	Y
1	60
2	55
3	80
4	75
2	90

Table 4

However, in table 4, there is no functional dependencies of Y on X, since two identical x values (x2 and x5) correspond to different y values. It is impossible to tell whether y will be 55 or 90 given x = 2.

We can use the below structure to examine functional dependencies in attributes.

If  $x_i = x_j \dots (1)$

then  $y_i = y_j \dots (2)$

i and j are random two indices (row numbers) in the table;  $i \neq j$ . If (1) stands, we must continue to check if (2) stands. However, if (1) doesn't stand, meaning that  $x_i \neq x_j$ , we don't even need to check (2). We can draw a conclusion that y is functionally dependent on x if all the values in the table satisfy the conditions listed above. That is, every unique x value determines a y value.

In the context of relational databases, a candidate key (or minimal super key) is the minimal set of attributes (columns) in a table that can be used to uniquely identify a row in the table. Candidate keys help to ensure that there are no duplicates or redundancies in the table. Every table has at least one candidate key. Candidate keys facilitate data retrieval by ensuring consistencies and relationships across different tables.

A candidate key must be the smallest possible combination of attributes that can ensure all unique entries in the table. It must contain unique and non-null values.

StudentID	Class	Name
1	A	Tom
2	A	Nick
3	B	Ann
4	C	Matt

Table 5

In table 5, it is possible to use StudentID and Name to uniquely identify each row of data, so StudentID and Name are both candidate keys. Class is not a candidate key because there are

duplicate values, A, and we are not able to identify an unique student given Class values. A set of two columns e.g., {StudentID, Class} also allow us to uniquely identify a single row. Yet, a candidate key must be the minimal set of columns, either StudentID or Name is sufficient as an identifier in this case. Therefore, in this case, StudentID and Name are the candidate keys but not the set of StudentID and Class.

Class	Gender	Name
A	M	Tom
A	M	Nick
B	F	Ann
C	M	Tom

Table 6

In table 6, it is impossible to identify a unique row of data using one column (attribute). However, the combination of Class and Name can serve as an identifier. For instance, although there are repetitive values in Class and Name, there is only one Tom in class A and C. Hence, the set of Class and Name i.e., {Class, Name} is the candidate key in this case. The set of Gender and Name cannot be the candidate key, as both Tom in Class A and C are all males, denoted by M in the table.

Primary key is one of the candidate keys within a table. It is in the subset of candidate key. The difference between primary key and candidate key is that there can only be one primary key to each table but many candidate keys. Same as a candidate key, a primary key must contain unique and non-null values and can be multiple or one single column in a table. Primary key serves as an unique identifier to search for a row in a table.

Class	Gender	Name	Score
A	M	Tom	90
A	M	Nick	90
B	F	Ann	80
C	M	Tom	70

Table 7

In table 7, {Class, Name}, {Name, Score} are candidate keys, but only one of them is chosen as primary key. The primary key in this case is a set of attributes.

Class	Gender	Name	Score
A	M	Tom	90
A	M	Nick	90
B	F	Ann	80
C	M	Ramond	70

Table 8

In table 8, a single attribute Name is sufficient to identify an unique row, so the primary key is therefore Name only.

## b)

A relational database table is in first normal form when it satisfies both the conditions of atomic values and no repeating values. Atomic values means every column should contain only one value that cannot be divided further. No repeating groups indicates that there are

no repeating columns that describing the same attributes. Here we use tables containing customer information to describe first normal form. A customer may stay in multiple cities and has multiple City values accordingly.

ID	Name	Age	Salary	City	Country
1	May	32	2000.00	Taipei, Kaohsiung	Taiwan
2	Mukesh	40	5000.00	New York	USA
3	Yuta	45	4500.00	Osaka	Japan

Table 9

Table 9 violates the condition of atomic values because it contains two City values in the first entry. Table 9 is therefore not in first normal form.

ID	Name	Age	Salary	City1	City2	Country
1	May	32	2000.00	Taipei	Kaohsiung	Taiwan
2	Mukesh	40	5000.00	New York		USA
3	Yuta	45	4500.00	Osaka		Japan

Table 10

Table 10 violates the condition of no repeating groups because column City1 and City2 both contain City values. The table is therefore not in first normal form.

A method to bring the table to first normal form to separate it into two tables. The repeating column, City, is moved to a table 12, and City values of the same Country are represented as rows. ID and Country are included in table 12 as foreign keys to link with table 11. The new tables conform with the conditions of atomic values and no repeating groups. We can include multiple City values for the same customer without wasting the space.

ID	Name	Age	Salary	Country
1	May	32	2000.00	Taiwan
2	Mukesh	40	5000.00	USA
3	Yuta	45	4500.00	Japan

Table 11

ID	City	Country
1	Taipei	Taiwan
1	Kaohsiung	Taiwan
2	New York	USA
3	Osaka	Japan

Table 12

c)

A relational database table is in second normal form when it is in first normal form and no partial dependencies. Full dependency means all non-prime attributes are fully functional

dependent on the primary key. Partial dependency means some non-prime attributes depends on only a subset of the primary key, when the primary is a composite. The tables below illustrate the concept.

CustomerID	StoreID	City
1	1	London
1	3	Manchester
2	1	London
3	2	Leeds
4	3	Manchester
5	4	London

*Table 13*

Table 13 is in first normal form since it satisfies atomic values, which no value can be divided further, and no repeating groups, which there are no two columns referring to the same category of values.

The primary key of the table is a composite, a set of CustomerID and StoreID, and City is the non-prime attribute. City is functionally dependent on StoreID, since each store can only locate in one city, while in a city there can be many stores. However, this also indicates that City only depends on part (or a subset) of the primary key, meaning that table 13 is not in second normal form.

To bring the table to second normal form, we split table 13 into two. City, which is the attribute partially dependent on the primary key, is moved to table 15, along with the StoreID, the attribute City is fully dependent on. The duplicates in table 15 are eliminated. The remaining attributes forms table 14. Now in table 15, City is fully functional dependent on StoreID, the primary key of the table. The two tables satisfy second normal form, as each of them is in first normal form and their non-prime attributes fully depend on its primary key.

CustomerID	StoreID
1	1
1	3
2	1
3	2
4	3
5	4

*Table 14*

StoreID	City
1	London
2	Leeds
3	Manchester
4	London

*Table 15*

#### **d)**

A relational database table is in third normal form when it is in second normal form and when there is no transitive functional dependency. Transitive functional dependency

describes the relationships of attributes in a table as the following. For instance, A, B C, are attributes of a table. A is functionally dependent on B, B is functionally dependent on C. Therefore, C is transitively dependent on A via B.

FilmID	GenreID	GenreType	Year
1	1	Horror	2012
2	2	Comedy	2010
3	1	Horror	2022
4	3	Animation	2021
5	2	Comedy	2003

Table 16

Table 16 satisfies first normal form, since all values are atomic and there are no repeating attributes. The primary key of table 16 is FilmID, and all non-prime attributes (Genre, GenreType, and Year) fully depends on FilmID, as each film must belong to a certain genre and has a release year. Table 16 thus satisfies second normal form. However, the dependencies of GenreType on FilmID is transitive. FilmID determines GenreID, and GenreID determines GenreType. Table 16 does not conform to third normal form.

FilmID	GenreID	Year
1	1	2012
2	2	2010
3	1	2022
4	3	2021
5	2	2003

Table 17

GenreID	GenreType
1	Horror
2	Comedy
3	Animation

Table 18

To bring table 16 to third normal form, we split it into table 17 and 18. GenreID and GenreType are moved table 18, as the latter fully depends on the former. Duplicates in table 18 are eliminated. The remaining attributes forms table 17. Now, in table 17. non-prime attributes, GenreID and Year are fully functional dependent only on its primary key, FilmID. In table 18, non-prime attributes, GenreType, is also fully dependent on only primary key, GenreID. The two tables conform to third normal form.

## Question 2

Open command prompt, launch PostgreSQL by typing “psql -U postgres” and inputting the password.

```
C:\Users\Cinthia Wei>psql -U postgres
Password for user postgres:
psql (16.0)
WARNING: Console code page (850) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
Type "help" for help.
```

*Screenshot 1*

```
postgres=# CREATE DATABASE new;
CREATE DATABASE
```

*Screenshot 2*

Inside postgres server, create a new database called “new” by typing “CREATE DATABASE new”. Note that command line is case insensitive, so the upper and lower cases are just customary practice to differentiate syntaxes and variable names.

Connect to the new database by typing “\c new”.

```
postgres=# \c new
You are now connected to database "new" as user "postgres".
new=#
```

*Screenshot 3*

Create a table called “employee” in the new database by the below codes. Here we specifies EmployeeID, FirstName, LastName, Gender to datatype VARCHAR(50), as the attributes will have to store string or a combination of string and numeric data later. BirthDate is DATE datatype, and Salary is INT (integer) datatype. The terminal will return “CREATE TABLE” once the table is successfully created.

```
new=# CREATE TABLE employee(
new(# EmployeeID VARCHAR(50),
new(# FirstName VARCHAR(50),
new(# LastName VARCHAR(50),
new(# Birthdate DATE,
new(# Gender VARCHAR(50),
new(# Salary INT
new(# );
CREATE TABLE
new=#
```

*Screenshot 4*

Insert values into employee table by the following code. In the first row of codes, specify the name of the table that we want to insert values in, and mention the attributes in the bracket. Be mindful that it is necessary to put non-numeric values inside quotation or double quotation mark. The inserted values have to strictly follow the datatype of the attributes. we can insert multiple rows of values at once by separating the brackets with comma.



```
new=# INSERT INTO employee(EmployeeID,FirstName,LastName,Birthdate,Gender,Salary)
new=# VALUES('E1001','John','Thomas','1976-09-01','M',100000),
new=# ('E1002','Alice','James','1972-07-31','F',80000),
new=# ('E1003','Steve','Wells','1980-10-08','M',50000),
new=# ('E1004','Santosh','Kumar','1985-07-20','M',60000),
new=# ('E1005','Ahmed','Hussain','1981-01-04','M',70000),
new=# ('E1006','Nancy','Allen','1978-05-05','F',90000);
INSERT 0 6
```

Screenshot 5

It is recommended to use “SELECT \* FROM employee” to call the table and double confirm that the table looks the way we want.

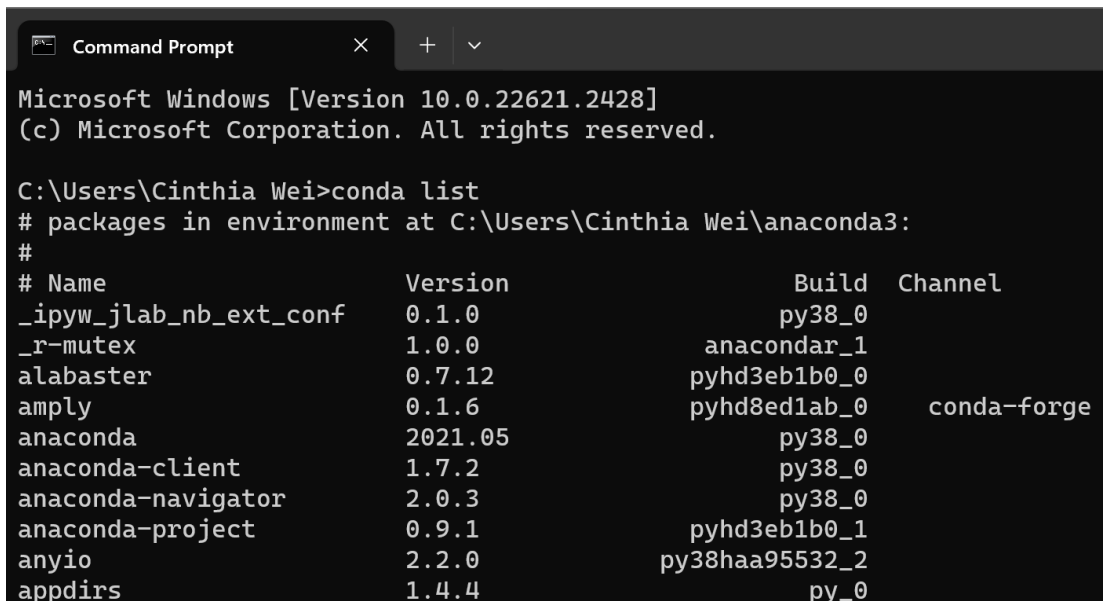
```
new=# SELECT * FROM employee;
employeeid | firstname | lastname | birthdate | gender | salary
-----+-----+-----+-----+-----+-----
E1001      | John     | Thomas  | 1976-09-01 | M      | 100000
E1002      | Alice    | James   | 1972-07-31 | F      | 80000
E1003      | Steve    | Wells   | 1980-10-08 | M      | 50000
E1004      | Santosh  | Kumar   | 1985-07-20 | M      | 60000
E1005      | Ahmed    | Hussain | 1981-01-04 | M      | 70000
E1006      | Nancy    | Allen   | 1978-05-05 | F      | 90000
(6 rows)
```

Screenshot 6

### Question 3

a)

To check whether Flask library is installed in Python, open command prompt and type “conda list”. If the terminal returns a list of all the libraries in anaconda3. Scroll down the terminal and it is found that Flask is already installed.



```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Cinthia Wei>conda list
# packages in environment at C:\Users\Cinthia Wei\anaconda3:
#
# Name                                Version                                Build      Channel
_ipyw_jlab_nb_ext_conf                0.1.0                                py38_0
_r-mutex                              1.0.0                                anacondar_1
alabaster                              0.7.12                               pyhd3eb1b0_0
amply                                  0.1.6                                pyhd8ed1ab_0
anaconda                              2021.05                               py38_0
anaconda-client                        1.7.2                                py38_0
anaconda-navigator                     2.0.3                                py38_0
anaconda-project                       0.9.1                                pyhd3eb1b0_1
anyio                                  2.2.0                                py38haa95532_2
appdirs                                1.4.4                                py_0
```

Screenshot 7

fastcache	1.1.0	py38he774522_0
filelock	3.0.12	pyhd3eb1b0_1
flake8	3.9.0	pyhd3eb1b0_0
flask	1.1.2	pyhd3eb1b0_0
freetype	2.10.4	hd328e21_0
fsspec	0.9.0	pyhd3eb1b0_0

Screenshot 8

b)

In Windows command prompt, change the directory to desktop by typing 'cd desktop'.

```
C:\Users\Cinthia Wei>cd desktop
C:\Users\Cinthia Wei\Desktop>
```

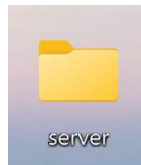
Screenshot 9

Create a folder named 'server' on the desktop by typing 'mkdir server', which 'mkdir' refers to 'make directory'.

```
C:\Users\Cinthia Wei\Desktop>mkdir server
```

Screenshot 10

Now we can see the folder on the desktop.



Screenshot 11

Navigate to the server folder by typing 'cd server'.

```
C:\Users\Cinthia Wei\Desktop>cd server
```

Screenshot 12

Use 'echo' to add contents, one line at a time, to the python file called 'app'.

```
C:\Users\Cinthia Wei\Desktop\server>echo from flask import Flask > app.py
C:\Users\Cinthia Wei\Desktop\server>echo app = Flask(__name__) >> app.py
C:\Users\Cinthia Wei\Desktop\server>echo @app.route('/') >> app.py
C:\Users\Cinthia Wei\Desktop\server>echo def hello_world(): >> app.py
C:\Users\Cinthia Wei\Desktop\server>echo     return 'Hello, World!' >> app.py
```

Screenshot 13

We can review if the file is successfully created by typing 'ls', which lists the contents of the current folder, or by opening the server folder on the desktop.

```

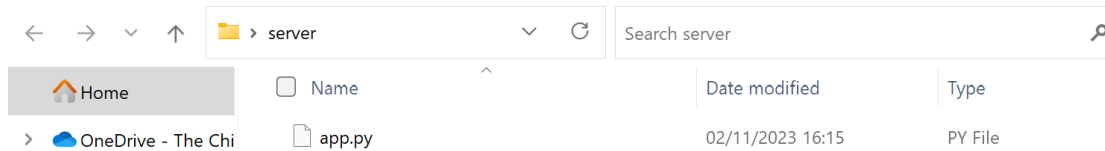
C:\Users\Cinthia Wei\Desktop\server>dir
Volume in drive C is OS
Volume Serial Number is 6684-1EA3

Directory of C:\Users\Cinthia Wei\Desktop\server

02/11/2023  16:15    <DIR>          .
02/11/2023  16:08    <DIR>          ..
02/11/2023  16:15                118 app.py
               1 File(s)                118 bytes
               2 Dir(s)  793,716,391,936 bytes free

```

Screenshot 14



Screenshot 15

In the same directory, type the below codes as screenshot 16. The first line 'set FLASK\_APP=app.py' sets the environment variable FLASK\_APP to the value 'app.py'. The second line sets the FLASK\_ENV environment variable to the value 'development'. Setting FLASK\_ENV to development enables debugging and provides more detailed error messages when running your Flask application. The third line 'FLASK\_DEBUG=True' enables Flask's debugging mode. 'FLASK\_DEBUG=1' works the same way. The final line 'python -m flask run' is used to run the Flask application in Python. Note that there should be no spaces on the two sides of equal when using 'set'. The terminal returns as shown in screenshot 16. We are now in a development environment and the debug mode is turned on.

```

C:\Users\Cinthia Wei\Desktop\server>set FLASK_APP=app.py

C:\Users\Cinthia Wei\Desktop\server>set FLASK_ENV=development

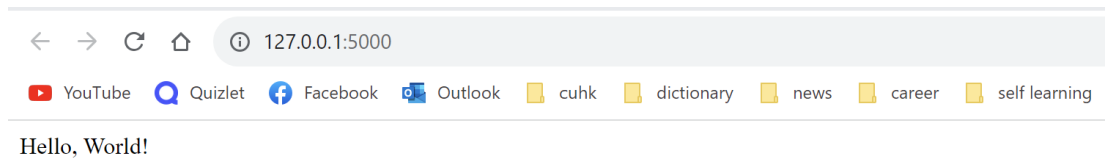
C:\Users\Cinthia Wei\Desktop\server>set FLASK_DEBUG=True

C:\Users\Cinthia Wei\Desktop\server>python -m flask run
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Restarting with windowsapi reloader
* Debugger is active!
* Debugger PIN: 321-805-559
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

```

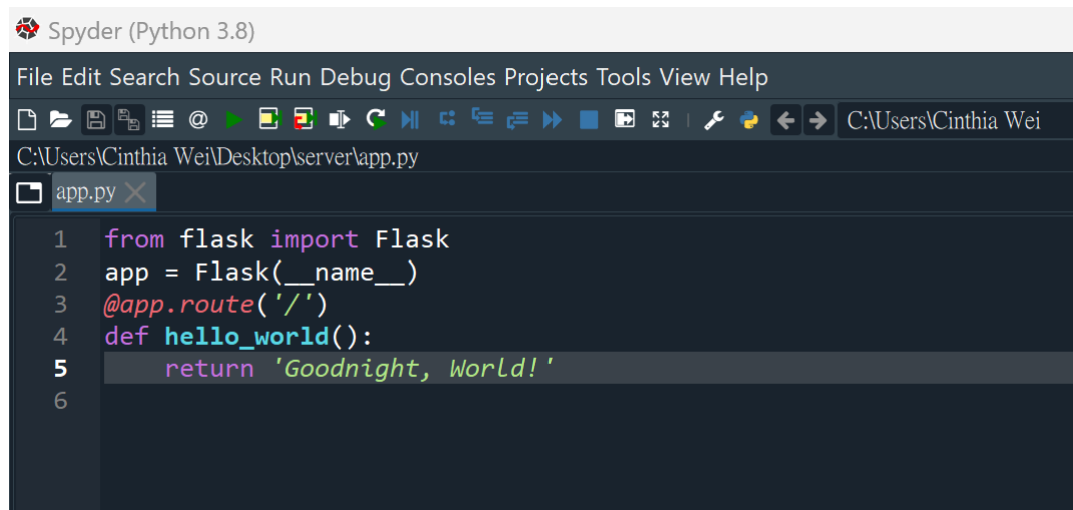
Screenshot 16

Ctrl and click on link '<http://127.0.0.1:5000/>', the webpage will pop out.



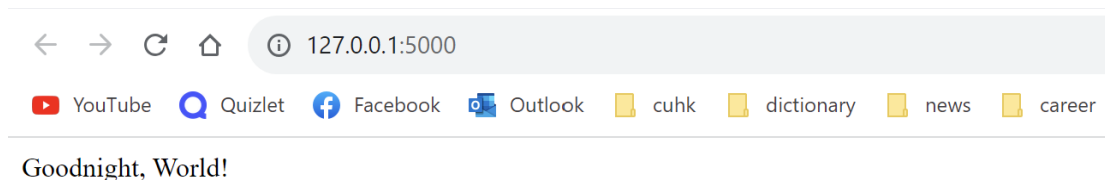
Screenshot 17

If we want to change the code while developing, simply open the text editor in your device. Spyder is used in the example. Open the app.py file in the text editor and change the return value of the codes. Save the file.



Screenshot 18

Go back to the webpage and refresh, the updated content should be displayed.



Screenshot 19

Every activity regarding the Flask application will be recorded on the terminal.

```

C:\Users\Cinthia Wei\Desktop\server>python -m flask run
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Restarting with windowsapi reloader
* Debugger is active!
* Debugger PIN: 321-805-559
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [05/Nov/2023 10:14:26] "GET / HTTP/1.1" 200 -
* Detected change in 'C:\\Users\\Cinthia Wei\\Desktop\\server\\app.py', reloading
* Detected change in 'C:\\Users\\Cinthia Wei\\Desktop\\server\\app.py', reloading
* Detected change in 'C:\\Users\\Cinthia Wei\\Desktop\\server\\app.py', reloading
* Restarting with windowsapi reloader
* Debugger is active!
* Debugger PIN: 321-805-559
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [05/Nov/2023 10:14:50] "GET / HTTP/1.1" 200 -

```

Screenshot 20

If there are spaces on the two side the equal sign when using 'set', we cannot turn on the debug mode. The updates on python file will only show up if we quit the and re-run the codes.

```

C:\Users\Cinthia Wei\Desktop\server>set FLASK_ENV = development
C:\Users\Cinthia Wei\Desktop\server>set FLASK_APP = app.py
C:\Users\Cinthia Wei\Desktop\server>set FLASK_DEBUG = True
C:\Users\Cinthia Wei\Desktop\server>python -m flask run
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

```

Screenshot 21

d)

Create a new folder called 'templates' inside the server folder. Change directory to templates. Create a file by using notepad text editor and name it 'index.html'.

```

C:\Users\Cinthia Wei\Desktop\server>mkdir templates
C:\Users\Cinthia Wei\Desktop\server>cd templates

```

Screenshot 22

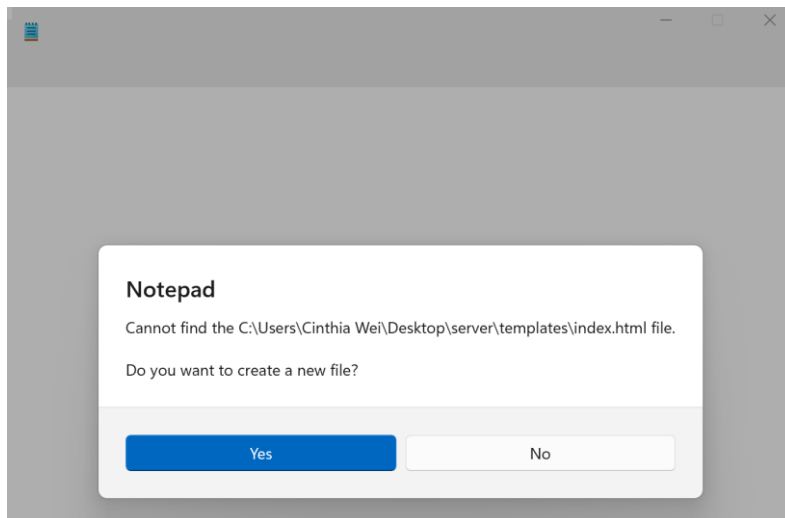
```

C:\Users\Cinthia Wei\Desktop\server\templates>mkdir notepad index.html

```

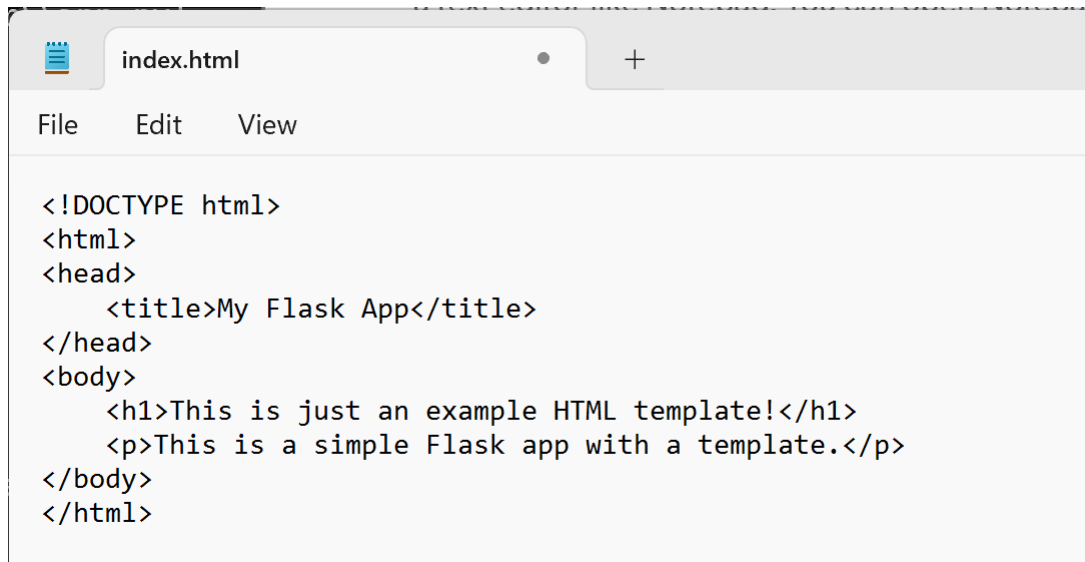
Screenshot 23

There will be pop up message asking whether you want to create a new file. Click yes to proceed.



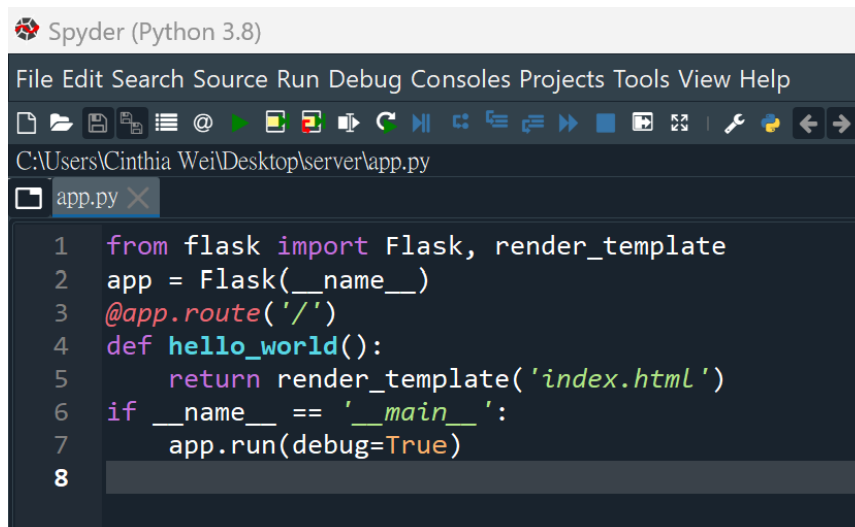
Screenshot 24

In notepad, type the below codes to set up the HTML template. Each tag starts with <> and ends with </>. Save the file and close notepad.



Screenshot 25

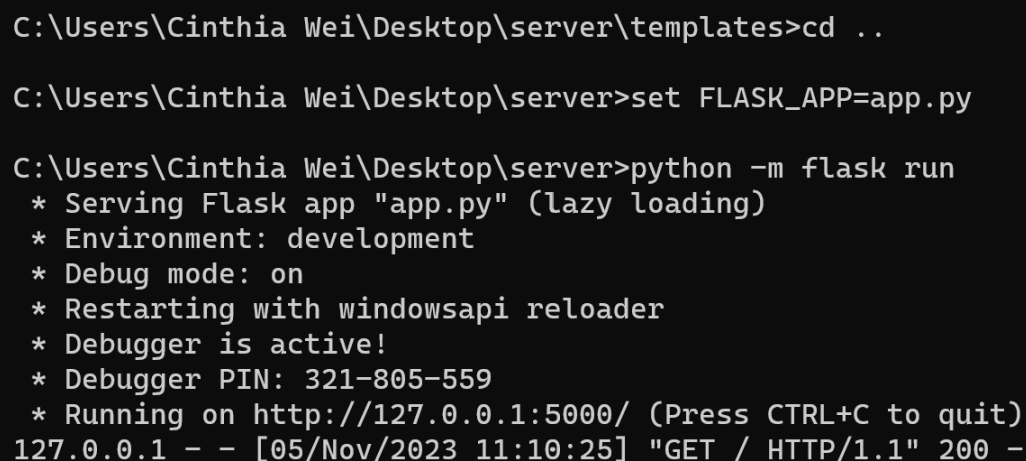
Open Spyder and make the following modifications on the codes. In the first line, we import the 'render\_template' function from the Flask library. 'render\_template('index.html')' tells Flask to look for the "index.html" file in the "templates" directory of app.py and render it. The render\_template function will take the "index.html" template, process it, and replace any dynamic content or placeholders in the template with the appropriate data. It allows you to generate dynamic HTML content by combining the template with data. We set the debug mode to True in order to see the updates automatically.

A screenshot of the Spyder Python IDE interface. The title bar says 'Spyder (Python 3.8)'. The menu bar includes 'File', 'Edit', 'Search', 'Source', 'Run', 'Debug', 'Consoles', 'Projects', 'Tools', 'View', and 'Help'. Below the menu bar is a toolbar with various icons. The file explorer on the left shows the file path 'C:\Users\Cinthia Wei\Desktop\server\app.py'. The main editor window displays the following Python code:

```
1 from flask import Flask, render_template
2 app = Flask(__name__)
3 @app.route('/')
4 def hello_world():
5     return render_template('index.html')
6 if __name__ == '__main__':
7     app.run(debug=True)
8
```

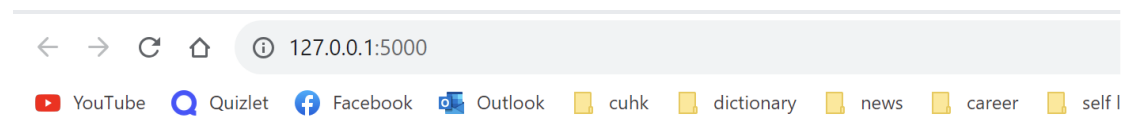
Screenshot 26

Navigate back to the server folder, the location of the app.py file. 'cd..' means going back to the previous folder level. Run the Flask application again and click on to the link. We can see the updated content. If the Flask application is running in development mode, simply refresh the web page and the updates will be shown.

A screenshot of a terminal window with a black background and white text. The commands and output are as follows:

```
C:\Users\Cinthia Wei\Desktop\server\templates>cd ..
C:\Users\Cinthia Wei\Desktop\server>set FLASK_APP=app.py
C:\Users\Cinthia Wei\Desktop\server>python -m flask run
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Restarting with windowsapi reloader
* Debugger is active!
* Debugger PIN: 321-805-559
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [05/Nov/2023 11:10:25] "GET / HTTP/1.1" 200 -
```

Screenshot 27



## This is just an example HTML template!

This is a simple Flask app with a template.

Screenshot 28

## Question 4

a)

The installation process varies from device to device. In the example, when trying to install Flask-SQLAlchemy via Windows Command Prompt, the following error occurs.

```
C:\Users\Cinthia Wei>conda install -c anaconda sqlalchemy
Collecting package metadata (current_repodata.json): done
Solving environment: -
The environment is inconsistent, please check the package plan carefully
The following packages are causing the inconsistency:

- defaults/noarch::alabaster==0.7.12=pyhd3eb1b0_0
- conda-forge/noarch::amply==0.1.6=pyhd8ed1ab_0
```

Screenshot 29

To solve the inconsistency in the conda environment, type 'y' to proceed when conda tried to solve the inconsistency. This may involve upgrading and downgrading some packages. If the method doesn't work, try typing 'conda update --all'. After finish updating, type 'conda list' to confirm that there is Flask-SQLAlchemy in the list of packages.

```
C:\Users\Cinthia Wei>conda list
# packages in environment at C:\Users\Cinthia Wei\anaconda3:
#
# Name                                Version                                Build      Channel
_anaconda_depends                    2023.09                               py38_mkl_1
_ipyw_jlab_nb_ext_conf              0.1.0                                 py38_0
_r-mutex                             1.0.0                                 anacondar_1
abseil-cpp                           20211102.0                           hd77b12b_0
aiobotocore                          2.5.0                                 py38haa95532_0
```

Screenshot 30

```
filelock                             3.0.12                               pyhd3eb1b0_1
flake8                               3.9.0                               pyhd3eb1b0_0
flask                                 2.2.2                               py38haa95532_0
flask-sqlalchemy                     3.0.2                               py38haa95532_0
freetype                             2.12.1                              ha860e81_0
frozenlist                           1.4.0                               py38h2bbff1b_0
fsspec                               2023.4.0                            py38haa95532_0
```

Screenshot 31

Once Flask-SQLAlchemy is installed, update the codes in the 'app.py' file. Open a text editor, Spyder is used in the example, the first two lines of codes indicate the import of required packages. In the next three lines, configure the Flask app to use SQLAlchemy and provide the database connection URI. The URL contains information including username, password, port number, and database name.

Use the class-object function to define a database model called Employee and link the model to the existing table called 'employee' in the database. The columns that will be shown on the web app are 'employeeid', 'firstname', 'lastname', 'birthdate', 'gender', 'salary'. The columns contains string, integer, or date values respectively.



'@app.route' defines the Flask route that retrieves data from the 'customer' table and renders it using a HTML template called 'list\_customers'.

```
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
# Configure the PostgreSQL database URI. Replace with your actual database URI.
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://postgres:cindy880928@localhost:5432/new'

# Initialize the SQLAlchemy extension
db = SQLAlchemy(app)

# Define the data model for your table
class Employee(db.Model):
    __tablename__ = 'employee'
    employeeid = db.Column(db.String(10), primary_key=True)
    firstname = db.Column(db.String(50))
    lastname = db.Column(db.String(50))
    birthdate = db.Column(db.Date)
    gender = db.Column(db.String(1))
    salary = db.Column(db.Integer)

# Create a route to display the data
@app.route('/')
def display_data():
    employees = Employee.query.all()
    return render_template('list_employees.html', employees=employees)

if __name__ == '__main__':
    app.run(debug=True)
```

Screenshot 32

Create a HTML template called 'list\_employees' in the 'templates' folder. 'templates' is at the same level as app.py. We can use text editor like spyder to edit the HTML template. Type the following codes in list\_employees. The <> refers to tags, and we can put contents such as texts or links inside tags. Each tag is closed by </>. The html template will look like screenshot 33. In <td> tags, we call each column of the employee table.

```

<!DOCTYPE html>
<html>
<head>
  <title>Employee List</title>
</head>
<body>
  <h1>Employee List</h1>
  <table>
    <thead>
      <tr>
        <th>Employee ID</th>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Birth Date</th>
        <th>Gender</th>
        <th>Salary</th>
      </tr>
    </thead>
    <tbody>
      {% for employee in employees %}
        <tr>
          <td>{{ employee.employeeid }}</td>
          <td>{{ employee.firstname }}</td>
          <td>{{ employee.lastname }}</td>
          <td>{{ employee.birthdate }}</td>
          <td>{{ employee.gender }}</td>
          <td>{{ employee.salary }}</td>
        </tr>
      {% endfor %}
    </tbody>
  </table>
</body>
</html>

```

Screenshot 33

Go back to command line and navigate to the server folder. Note that command prompt is case insensitive. Call the Flask app and run it. It is suggested to turn on the debug mode to see the updates easily.

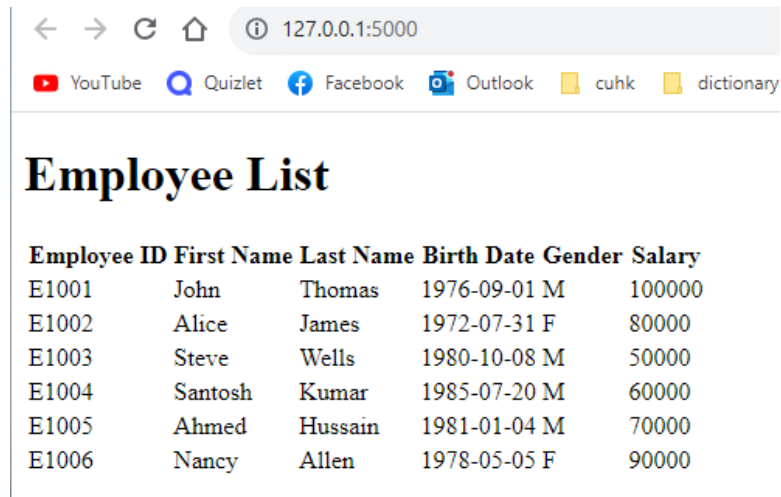
```

C:\Users\Cinthia Wei>cd desktop
C:\Users\Cinthia Wei\Desktop>cd server
C:\Users\Cinthia Wei\Desktop\server>set flask_app=app.py
C:\Users\Cinthia Wei\Desktop\server>set flask_env=development
C:\Users\Cinthia Wei\Desktop\server>set flask_debug=1
C:\Users\Cinthia Wei\Desktop\server>python -m flask run
* Serving Flask app 'app.py'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit

```

Screenshot 34

Click on to the link and the web page will be show the list of employee information based on employee table in new database.



Employee ID	First Name	Last Name	Birth Date	Gender	Salary
E1001	John	Thomas	1976-09-01	M	100000
E1002	Alice	James	1972-07-31	F	80000
E1003	Steve	Wells	1980-10-08	M	50000
E1004	Santosh	Kumar	1985-07-20	M	60000
E1005	Ahmed	Hussain	1981-01-04	M	70000
E1006	Nancy	Allen	1978-05-05	F	90000

Screenshot 35

b)

To show the total number of rows in the 'employee' table, update the app.py codes as screenshot 36. In particular, modify the route to add another query 'Employee.query.count()'. Assign the query to variable 'total\_rows' and return the new variable accordingly.

```
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
# Configure the PostgreSQL database URI. Replace with your actual database URI.
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://postgres:cindy880928@localhost:5432/new'

# Initialize the SQLAlchemy extension
db = SQLAlchemy(app)

# Define the data model for your table
class Employee(db.Model):
    __tablename__ = 'employee'
    employeeid = db.Column(db.String(10), primary_key=True)
    firstname = db.Column(db.String(50))
    lastname = db.Column(db.String(50))
    birthdate = db.Column(db.Date)
    gender = db.Column(db.String(1))
    salary = db.Column(db.Integer)

# Create a route to display the data
@app.route('/')
def display_data():
    employees = Employee.query.all()
    # Count the total number of rows in the table
    total_rows = Employee.query.count()
    return render_template('list_employees.html', employees=employees, total_rows=total_rows)

if __name__ == '__main__':
    app.run(debug=True)
```

Screenshot 36

Update the list\_customer.html to show the result of the new query. Add a new set of tags <p> </p> and insert total\_rows.

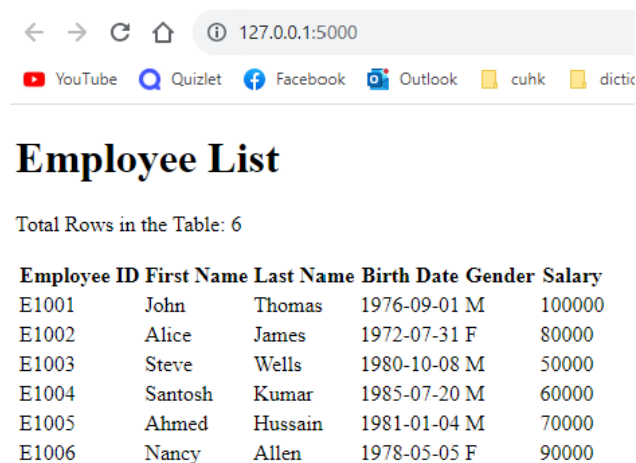
```

<!DOCTYPE html>
<html>
<head>
  <title>Employee List</title>
</head>
<body>
  <h1>Employee List</h1>
  <!-- Display the total number of rows -->
  <p>Total Rows in the Table: {{ total_rows }}</p>
  <table>
    <thead>
      <tr>
        <th>Employee ID</th>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Birth Date</th>
        <th>Gender</th>
        <th>Salary</th>
      </tr>
    </thead>
    <tbody>
      {% for employee in employees %}
        <tr>
          <td>{{ employee.employeeid }}</td>
          <td>{{ employee.firstname }}</td>
          <td>{{ employee.lastname }}</td>
          <td>{{ employee.birthdate }}</td>
          <td>{{ employee.gender }}</td>
          <td>{{ employee.salary }}</td>
        </tr>
      {% endfor %}
    </tbody>
  </table>
</body>
</html>

```

Screenshot 37

Save the files and refresh the web page. The total rows in the table is successfully added.



Employee List

Total Rows in the Table: 6

Employee ID	First Name	Last Name	Birth Date	Gender	Salary
E1001	John	Thomas	1976-09-01	M	100000
E1002	Alice	James	1972-07-31	F	80000
E1003	Steve	Wells	1980-10-08	M	50000
E1004	Santosh	Kumar	1985-07-20	M	60000
E1005	Ahmed	Hussain	1981-01-04	M	70000
E1006	Nancy	Allen	1978-05-05	F	90000

Screenshot 38

## Question 5

To add styles to the webpage, firstly, create a folder called 'static' at the same level of app.py. Create a .css file called 'style' inside the static folder. The style.css will be the stylesheet for the list\_employees.html.

Add the following codes to the style.css. Here we set the background to gray, align the html content to the middle top, and change the color and font of the table text. A white border is added to the table.

```
/* Set the background color to gray */
body {
    background-color: gray;
    display: flex;
    flex-direction: column;
    justify-content: flex-start;
    align-items: center;
    height: 100vh;
}

/* Set the title font color to pink */
h1 {
    color: pink;
}

/* Set column titles inside <th> to be yellow */
th {
    background-color: black;
    color: pink; /* Text color for column titles */
}

/* Set content font color to white */
p, td {
    color: white;
}

/* Set the font to Candara */
body {
    font-family: 'Candara', sans-serif;
}

/* Add a border to the table */
table {
    border: 2px solid white;
}
```

Screenshot 39

In app.py, update a new line of codes as screenshot 40, which tells flask to look into the static folder.

```
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.static_folder = 'static' # Set the static folder
# Configure the PostgreSQL database URI. Replace with your actual database URI.
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://postgres:cindy880928@localhost:5432/new'
```

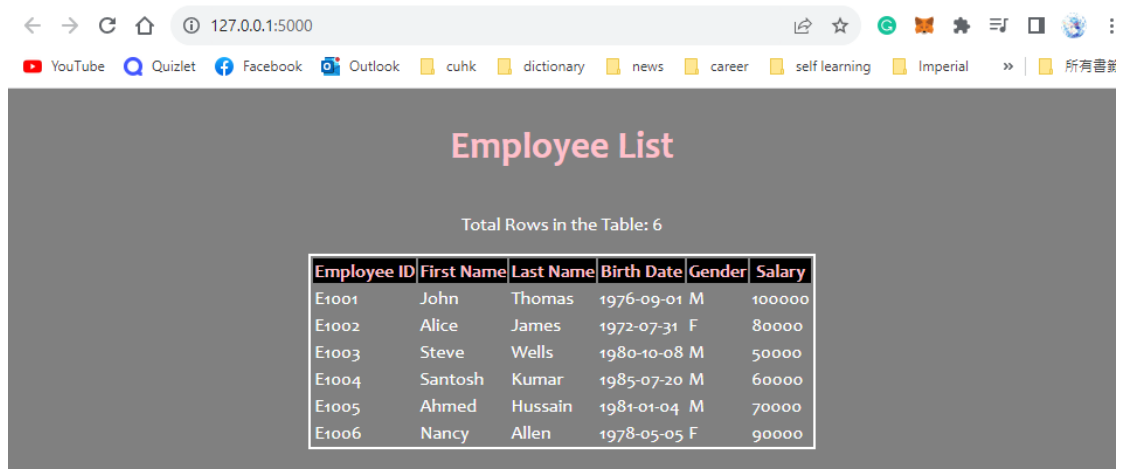
Screenshot 40

In list\_employees.html, add a new line of codes under <title>, as shown in screenshot 41. This links the html template to the stylesheet. After 'href', you can also use the relative path or absolute path of the stylesheet if the aforementioned steps don't work.

```
<!DOCTYPE html>
<html>
<head>
  <title>Employee List</title>
  <!-- Link to the CSS stylesheet -->
  <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css') }}" />
</head>
<body>
```

Screenshot 41

Save all the files and refresh the webpage, we will be able to see the updated visuals.



Employee ID	First Name	Last Name	Birth Date	Gender	Salary
E1001	John	Thomas	1976-09-01	M	100000
E1002	Alice	James	1972-07-31	F	80000
E1003	Steve	Wells	1980-10-08	M	50000
E1004	Santosh	Kumar	1985-07-20	M	60000
E1005	Ahmed	Hussain	1981-01-04	M	70000
E1006	Nancy	Allen	1978-05-05	F	90000

Screenshot 42

Note that the folder structure must be the same as screenshot 43 and 44 in order to prevent errors. If the style.css was in different location, the browser may not be able to find it and there will be 404 error.

Desktop > server > templates

Name	Date modified	Type	Size
static	07/11/2023 22:23	File folder	
list_employees.html	08/11/2023 22:52	Chrome HTML Docu...	2 KB
index.html	05/11/2023 10:56	Chrome HTML Docu...	1 KB

Screenshot 43

Desktop > server > templates > static

Name	Date modified	Type	Size
style.css	07/11/2023 11:44	Cascading Style Shee...	1 KB

Screenshot 44

## Question 6

To enable users to add new employee information to the table, firstly, create another html template called 'add\_employee'. In add\_employee.html, write the codes in accordance with screenshot 45. This html is also linked to the same stylesheet, as mentioned in <link>. In <form>, ensure that the form action attribute is set to '/add\_employee'. This update corresponds to the updates in app.py and will be mentioned in the following.

<input> creates input fields for users to type in the information of the new employee, in consistent with the datatype in each field. <label> above each <input> adds a text label to the input fields. A submit button is also added to the bottom of the webpage, which is specified in <button>.

```
<!DOCTYPE html>
<html>
<head>
  <title>Add Employee</title>
  <!-- Link to the CSS stylesheet -->
  <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css') }}" />
</head>
<body class="add-employee">
  <h1>Add Employee</h1>
  <form method="POST" action="/add_employee">
    <label for="employeeid">Employee ID:</label>
    <input type="text" id="employeeid" name="employeeid" required><br><br>

    <label for="firstname">First Name:</label>
    <input type="text" id="firstname" name="firstname" required><br><br>

    <label for="lastname">Last Name:</label>
    <input type="text" id="lastname" name="lastname" required><br><br>

    <label for="birthdate">Birthdate:</label>
    <input type="date" id="birthdate" name="birthdate" required lang="en"><br><br>

    <label for="gender">Gender:</label>
    <input type="text" id="gender" name="gender" required><br><br>

    <label for="salary">Salary:</label>
    <input type="number" id="salary" name="salary" required><br><br>
    <button type="submit">Submit</button>
  </form>
</body>
</html>
```

Screenshot 45

In list\_employees.html, update a new line of codes at the bottom to add an 'add employee' button, as specified in <button>.

```
</tbody>
</table>
<button type="button" class="add-employee-button" onclick="location.href='/add_employee'">Add Employee</button>
</body>
</html>
```

Screenshot 46

Some parts of app.py also require updates. Firstly, import request and redirect library.

```

from flask import Flask, render_template, request, redirect
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.static_folder = 'static' # Set the static folder
# Configure the PostgreSQL database URI. Replace with your actual database URI.
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://postgres:cindy880928@localhost:5432/new'

```

Screenshot 47

Define a new route under the first route, in accordance with screenshot 48. Define a new function that appends the new employee information to the database. Use `redirect()` to navigate to the `list_employees` route.

```

# Create a route to display the data
@app.route('/')
def display_data():
    employees = Employee.query.all()
    # Count the total number of rows in the table
    total_rows = Employee.query.count()
    return render_template('list_employees.html', employees=employees, total_rows=total_rows)

# New route for adding employees
@app.route('/add_employee', methods=['GET', 'POST'])
def add_employee():
    if request.method == 'POST':
        employeeid = request.form['employeeid']
        firstname = request.form['firstname']
        lastname = request.form['lastname']
        birthdate = request.form['birthdate']
        gender = request.form['gender']
        salary = request.form['salary']

        # Create a new Employee object and add it to the database
        new_employee = Employee(
            employeeid=employeeid,
            firstname=firstname,
            lastname=lastname,
            birthdate=birthdate,
            gender=gender,
            salary=salary
        )
        db.session.add(new_employee)
        db.session.commit()

        return redirect('/') # Redirect back to the list of employees

    return render_template('add_employee.html')

if __name__ == '__main__':
    app.run(debug=True)

```

Screenshot 48

In `style.css`, it is suggested to add the following codes to prettify the content style of `add_employee.html`. We can add it on the bottom of `style.css`.

```

/* Style the body text for the add_employee page */
body.add-employee {
    color: white;
}

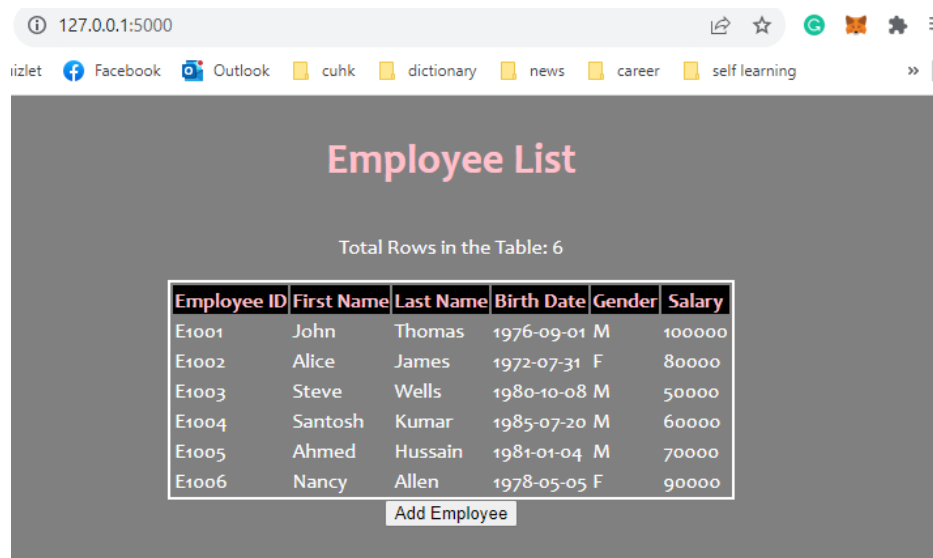
```

Screenshot 49

Save all the files and refresh the webpage, and we will be able to see the new button below



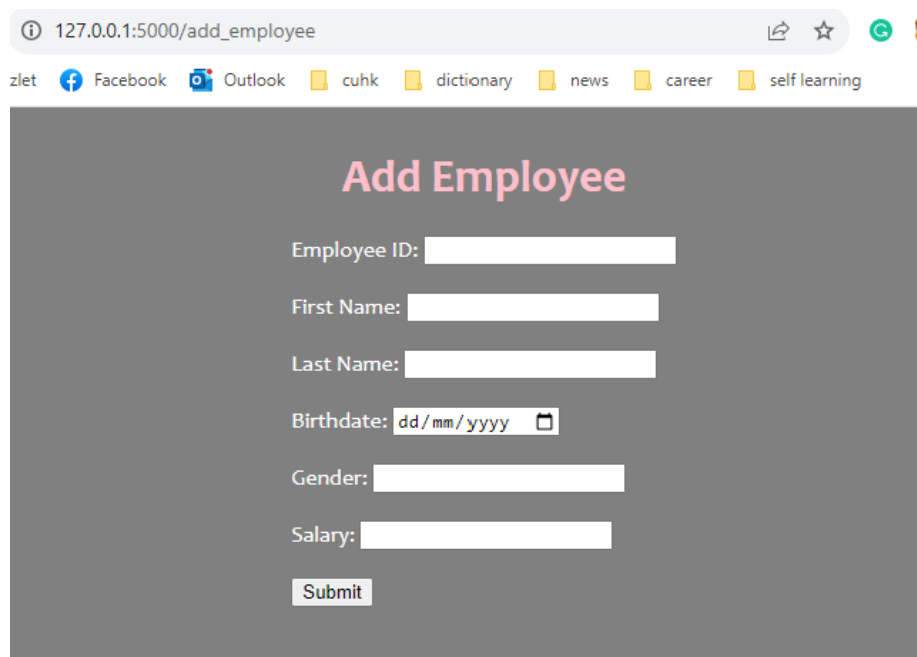
the table.



Employee ID	First Name	Last Name	Birth Date	Gender	Salary
E1001	John	Thomas	1976-09-01	M	100000
E1002	Alice	James	1972-07-31	F	80000
E1003	Steve	Wells	1980-10-08	M	50000
E1004	Santosh	Kumar	1985-07-20	M	60000
E1005	Ahmed	Hussain	1981-01-04	M	70000
E1006	Nancy	Allen	1978-05-05	F	90000

Screenshot 50

Click onto “Add Employee”, we will be directed to add\_employee.html, which contains input fields and has the same style as list\_employees.html.




**Add Employee**

Employee ID:

First Name:

Last Name:

Birthdate:  

Gender:

Salary:

Screenshot 51

Input the information about a new employee.

127.0.0.1:5000/add\_employee

zlet Facebook Outlook cuhk dictionary news career self learning

## Add Employee

Employee ID:

First Name:

Last Name:

Birthdate:

Gender:

Salary:

Screenshot 52

We will be auto directed to list\_employee.html. The table and total rows of the table will be updated accordingly.

127.0.0.1:5000

zlet Facebook Outlook cuhk dictionary news career self learning

## Employee List

Total Rows in the Table: 7

Employee ID	First Name	Last Name	Birth Date	Gender	Salary
E1001	John	Thomas	1976-09-01	M	100000
E1002	Alice	James	1972-07-31	F	80000
E1003	Steve	Wells	1980-10-08	M	50000
E1004	Santosh	Kumar	1985-07-20	M	60000
E1005	Ahmed	Hussain	1981-01-04	M	70000
E1006	Nancy	Allen	1978-05-05	F	90000
E1007	Kathy	Wu	1999-03-30	F	65000

Screenshot 53

To make sure that the newly added information is also updated in the table inside the database, use command prompt to launch PostgreSQL. Connect to the new database and query the employee table. There are now 7 rows of entries in the table, which is in accordance with the one on the webpage. The updates are successful in the database as well.

```

C:\Users\Cinthia Wei>psql -U postgres
Password for user postgres:
psql (16.0)
WARNING: Console code page (850) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \c new
You are now connected to database "new" as user "postgres".
new=# select * from employee;
 employeeid | firstname | lastname | birthdate | gender | salary
-----+-----+-----+-----+-----+-----
 E1001      | John     | Thomas  | 1976-09-01 | M      | 100000
 E1002      | Alice    | James   | 1972-07-31 | F      | 80000
 E1003      | Steve    | Wells   | 1980-10-08 | M      | 50000
 E1004      | Santosh  | Kumar   | 1985-07-20 | M      | 60000
 E1005      | Ahmed    | Hussain | 1981-01-04 | M      | 70000
 E1006      | Nancy    | Allen   | 1978-05-05 | F      | 90000
 E1007      | Kathy    | Wu      | 1999-03-30 | F      | 65000
(7 rows)

```

Screenshot 54

## Question 7

You are a consultant in analytics. Describe a situation in which a business could use a web app, backed by a relational database, to aid its operations; then propose a plan to deliver this solution. Your answer should discuss the current situation (before introducing the web app), motivation for the new solution, an implementation strategy, potential problems, and costings.

**Title:** How Web Application and Database Fuel The Business of HobbyConnect

### 1. Scenario Overview:

There is business named "HobbyConnect" that specializes in helping people find and connect with others who share their hobbies and interests. Currently, HobbyConnect operates primarily through social media groups and forums, allowing users to join discussions, share experiences, and meet like-minded hobbyists. However, they face challenges related to content moderation, user engagement, and data management. HobbyConnect aims to develop a web app backed by a relational database to streamline these operations, improve user experience, and enhance content control.

### 2. Current Situation:

Users currently rely on various social media platforms such as Discord, Facebook, and Instagram to connect with others who share their interests. However, these platforms have limitations in terms of content format, user management, and personalized experiences. For example, the user interface in Discord is not straightforward for many people, which hinders the growth in new users. Due to popular live-streaming feature of Discord, gaming is the most popular interest shared by users on the platform. Yet, having only one dominating topic on Discord is not an ideal situation, and HobbyConnect intends to include and activate other hobby groups on Discord. Since the business operates on three social media platforms

for the purpose of complementing their strengths and weaknesses, the user data management is overlapping, inconsistent, and incoherent, posing difficulties in drawing user behavioral insights. In terms of personalized experience, users utilize channels, key words to search for specific topics or hobby groups, but the searching experience is ineffective given the content structure of each social media. The current business model requires users to initiate the search themselves and fails to create personalized recommendation or record previous search.

HobbyConnect faces challenges related to content moderation, as it can be time-consuming to identify and remove harassing or inappropriate content from discussions. Since users primarily use private chat (both one-on-one and group) on Instagram and Discord to engage in fellow hobbyists, HobbyConnect can only indirectly rely on content moderation from these platforms, which is often not enough to protect their users. In the case that a user uses spams or use abusive or violent language, HobbyConnect addresses the problem only when staffs or moderators report it or when it receives tickets from user report system. HobbyConnect then removes or block the user manually, which is a passive and inefficient method to deal with wrongful behaviors from the users. Poor content moderation affects the overall quality of interactions on the platform.

### **3. Motivation for the New Solution:**

HobbyConnect wants to provide users with a more customized, seamless, and engaging experience where they can easily connect with others who share their hobbies. For instance, the company aims to offer users easier navigation to their ideal hobby groups and chat rooms by a filter or improved user interface in any forms, instead of scrolling through long list of posts on Facebook and channels on Discord. The company also plans to recommendations on hobby groups or topics they can participate in, based on their historical search and information on user account.

In addition, the new solution aims to implement more efficient content moderation algorithms to maintain a safe and respectful environment for all users. HobbyConnect aims to elevate and reorganize user data collection as a resource of machine learning, know-your-customers, and sentiment analysis. The company expects to use automation to identify toxic content and remove problematic users, ensuring a faster and less laborious content moderation process.

Lastly, HobbyConnect wants to extend the features on their services, for instance, allowing users to customized more about their profiles, add hobby tags, which can potentially improve user engagement and help them to discover discussions and group chats with like-minded hobbyists.

### **4. Implementation Strategy:**

The web app will be developed using the Flask web framework, a popular choice for building web applications in Python. Flask provides a lightweight yet powerful foundation for creating web applications. It offers a clean and simple API for handling HTTP requests, rendering templates, and more.

PostgreSQL will be used as the relational database for storing user data, hobby groups,

discussions, and content moderation logs, and so on. Flask-SQLAlchemy, an extension for Flask, will be employed to facilitate the interaction between the Flask application and the PostgreSQL database. It provides an Object Relational Mapping (ORM) layer for database operations.

On the aspect of user data collection and account creation, users will land on the web app's homepage and create user accounts on their first login. When users register, the web app will collect and store essential user data, such as usernames, email addresses, and passwords. Additional profile information, such as hobbies, interests, and preferred discussion topics, will be gathered as users complete their profiles. The database will store not only user data collected from user profile but also user-generated content based on search activities, clicks, views, and posts.

With the amount of valuable user data, the PostgreSQL database schema helps to manage the data by splitting it to the following tables:

`users`: To store user account information.

`hobbies`: To store user-defined hobbies and interests.

`groups`: To store hobby groups created by users.

`discussions`: To store discussion threads within hobby groups.

`messages`: To store user messages in discussions.

`moderation\_logs`: To log moderation actions for content control.

`user\_activity`: To track user activities and interactions.

Following, the web app will use Flask-SQLAlchemy to interact with the PostgreSQL database. It will create, read, update, and delete records in the tables. The web app will query the hobbies and groups tables to provide users with options for finding and joining hobby groups and topics.

## **5. Viable Benefits of the Implementation**

The implementation of a web app backed by a relational database can significantly help HobbyConnect generate valuable business insights by providing a robust foundation for data collection, storage, and analysis. For instance, user interactions with the web app can contribute to user behavioral insights, revealing popular hobbies or trending topics. In addition, data from user profile can provide demographic insights which help tailor marketing campaigns and target specific user segments.

Apart from that, the web app will include filters and search options which allow easier navigation for users and simplified process to discover hobby groups and topics that match their interests. Users can filter by interests, locations, group size, and activity level. Personalized recommendations will be generated based on users' previous searches, activities, and interactions within the app. Machine learning algorithms can analyze user data to provide tailored suggestions.

Furthermore, the development of web app contributes to better content moderation. With increasing user data, the machine learning model could be more accurate on identifying wrongful speech and behaviors of users or even predicting them based on certain indicators.

Users who violate the platform rule could be immediately addressed. In short, Automated content moderation algorithms will help maintain a respectful environment by detecting and flagging inappropriate content, enhancing user engagement and stickiness to HobbyConnect.

## 6. Potential Problems

HobbyConnect will have to address concerns related to user privacy and data security, ensuring that user data is stored securely and used responsibly. Terms and conditions of the use of user data should be stated clear to prevent legal issues, and rules on speech should be explained.

Despite the efficiency in content moderation brought by machine learning technologies, HobbyConnect has to be cautious about the potential risks of inaccuracy and bias in detecting harassing content and discrimination against certain demographics. Since machine learning models learn from historical data, and if the training data contains biases or presents poorly in certain languages and cultures, the models may inherit those biases. For example, if the training data has a disproportionate amount of toxic content targeting specific races, the model might over-identify toxic content within that group. Or, the model may mistakenly flag content that is innocuous but in a less commonly used language.

## 7. Costings

The below is a rough estimate of the cost involved in development of the web app and implementation of the database system. The estimates are based on average salary level in London as well as marketing, legal, software services in 2023.

**Development Costs:** costs for hiring web developers, designers, and data scientists for initial development and ongoing maintenance. Recruiting a web developer, either front-end or back-end costs around £41,000 per year per developer. Recruiting a UX/UI designer costs around £51,000 per year per designer. Recruiting a data scientist costs £59,000 per year per data scientist.

**Database Hosting:** Costs associated with PostgreSQL database hosting services. PostgreSQL is an open-source database, so there is no direct licensing cost. However, HobbyConnect may need to consider cloud hosting expenses which estimates at £100 per month.

**Domain and SSL:** Expenses for purchasing and renewing the domain name and obtaining SSL certificates for security. Domain registration costs around from £30 per year. SSL certificates can cost around £125 per year.

**Maintenance and Support:** Ongoing costs for platform maintenance, updates, and customer support. Ongoing maintenance and updates will be conducted by the development team. We assume that HobbyConnect's current workforce can cover the customer support services, so maintenance and support costs refer to the monthly salaries of the development team.

**Marketing:** Marketing costs can vary widely based on the scope and strategy. We assume HobbyConnect do not need to hire additional marketers. Costs incurred are mainly for buying online ads (e.g., YouTube and Instagram ads), which is budget at £1,500 per month.

**Legal and Compliance:** Expenses related to legal services for privacy policies, terms of service, and compliance with data protection regulations may cost around £35,000 per year

As a rough estimate, considering a team of 2 web developers, 2 UX/UI designers, 1 data scientist, along with cloud hosting, 3-month marketing efforts, domain and SSL, legal and compliance, the initial development costs approximate at £232,855. Ongoing monthly expenses for hosting, maintenance, and marketing could be around £17,600.

## **8. Conclusion**

By implementing this solution, HobbyConnect can enhance user experiences, efficiently moderate content, and foster a vibrant community of hobbyists, ultimately improving the success of their business.

## **Conclusion**

As explained in question 1 of the technical report, functional dependency, keys and different levels of normal forms which are commonly used to describe the relationships of data and tables in a relational database. If tables are in higher level of normal forms, data storing, updating, and retrieval is more efficient. From question 2 to 6, we walk through the development of a web application backed by PostgreSQL database and enable updates made on the affiliated web page to be linked to the database. In the development process, Flask provide the framework of building the web app, and Flask-SQLAlchemy allows interactions of Flask and database. The app.py file is the main application script, which is executed in Command Prompt. Several html templates define the structure of the web pages, which are binding with app.py. These components contribute to the development and functionality of the web app. Question 7 illustrates the usages, implementation, pros and cons of developing a web app for a hobby connection business called HobbyConnect. With a web app, HobbyConnect is able to improve data management, user experience, content moderation, and many other aspects of business. Yet, it is worth noting that developing a web app requires time and investment, and it may bring along user privacy issues and inaccurate business insights.

The standard procedure to troubleshoot when I confront with errors surrounds double-checking the codes myself, searching for solutions on Stack Overflow or Youtube, and reading documentations of the related languages. In most cases, I was able to find the information I need on Stack Overflow. Trial and error approach plays an important role especially when setting the environment and building the web page, and I found it convenient that the development and debug mode of flask allows me to track the activities of the app and see immediate updates on the web page whenever I modified the codes.

Hopefully, the technical report could provide useful information about front-end development and relational database for the users.