INTE2047 Information Systems Solutions and Design

Assessment 2 – Milestone 2

Best Practices

Date: 06/05/2022
Student ID: S3741327
Name: Juyeon Kim

Table of Contents

• Database

a. Scalable database design decision

Aligning with the need of Peters, a table named “Room Type” was created as in Figure 1 for the sake of scalability to handle additional types of room at a future date. As the business scales up and when there are numerous rooms needed to be added, this table will allow users to easily choose the type of room and create a room record.

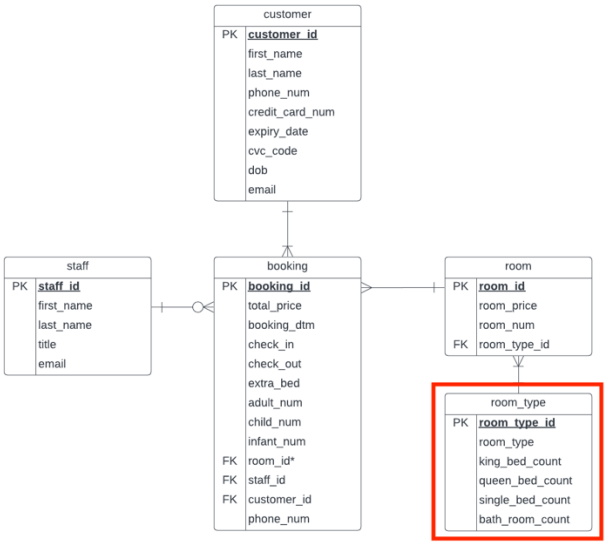

Figure 1. AHS ERD diagram

```
ROOM_TYPE_SQL = """
   CREATE TABLE room_type (
      room_type_id              INTEGER PRIMARY KEY AUTOINCREMENT,
      room_type_tlt             VARCHAR(50) NOT NULL,
      king_bed_count            INTEGER NOT NULL,
      queen_bed_count           INTEGER NOT NULL,
      single_bed_count          INTEGER NOT NULL,
      bath_room_count           INTEGER NOT NULL
)
"""
```

Code 1. Code snippet for creating room_type table

b. Selective application of “Delete on cascade”

"On delete cascade constraint” was being used in designing database to automatically delete the records from the child table when the records from the parent table are deleted. However, not all of referring key take this constraint. A room record will be deleted along with a room type record since the room rows are a logical extension of a room type and it is clear that if user want to delete all Delux type of rooms when deleting Delux from available room types. On

the other hand, DELETE CUSTOMER should not delete the associated BOOKINGs because BOOKINGs are not just attributes of a customer and it's important in their own.

```
ROOM_SQL = """
   CREATE TABLE room (
      room_id                INTEGER PRIMARY KEY AUTOINCREMENT,
      room_price              DOUBLE NOT NULL,
      room_num                INTEGER NOT NULL,
      room_type_id            INTEGER NOT NULL,
      FOREIGN KEY(room_type_id)  REFERENCES room_type(room_type_id) ON DELETE CASCADE
)
"""

BOOKING_SQL = """
   CREATE TABLE booking (
      booking_id             INTEGER PRIMARY KEY AUTOINCREMENT,
      check_in               TEXT,
      check_out              TEXT,
      extra_bed              BOOLEAN NOT NULL,
      adult_num              INTEGER NOT NULL,
      child_num              INTEGER NOT NULL,
      infant_num             INTEGER NOT NULL,
      total_price            DOUBLE NOT NULL,
      booking_dtm            TEXT,
      room_id                INTEGER NOT NULL,
      staff_id               INTEGER NOT NULL,
      customer_id            INTEGER NOT NULL,
      phone_num              VARCHAR(50) NOT NULL,
      FOREIGN KEY(room_id)     REFERENCES room(room_id)
      FOREIGN KEY(staff_id)    REFERENCES staff(staff_id)
      FOREIGN KEY(customer_id) REFERENCES customer(customer_id) ON DELETE CASCADE
)
"""
```

Code 2. ON DELETE CASCADE

c. Unique constraint

As email and phone numbers could be a unique identifier of each customer and staff, the unique constraint was applied to the email field of the customer and staff table and phone number of the customer table to prevent the same user creating duplicated records.

```
CUSTOMER_SQL = """
   CREATE TABLE customer (
      customer_id          INTEGER PRIMARY KEY AUTOINCREMENT,
      first_name           VARCHAR(50) NOT NULL,
      lastname             VARCHAR(50) NOT NULL,
      phone_num            VARCHAR(50) NOT NULL UNIQUE,
      email                VARCHAR(50) NOT NULL UNIQUE,
      credit_card_num      VARCHAR(50) NOT NULL,
      expiry_date          VARCHAR(50) NOT NULL,
      cvc_code             VARCHAR(50) NOT NULL,
      dob                  VARCHAR(50) NOT NULL
)
"""
```

Code 3. Unique constraint

- # GUI

a. Give user a chance to cancel an important operation

The system is designed to pop up confirmation message before deleting every record on a database to prevent a case where a user can accidentally delete any type of important record.

```
def confirm_delete(self):
    """
    Show confirmation popup before executing delete.

    Parameters: None

    Return: None
    """
    answer = askyesno(title='confirmation',
                message=f'Are you sure that you want to delete?')
    if answer:
        self.delete()
```

Code 4. Confirmation popup for record deletion

b. Various widget

The system takes various types of widgets by type of fields for better & intuitive user experience and to get rid of manual value typing. As in Code 5.1-3, the List box was used to present a list of records from which one can be selected. The combo box was designed to select a foreign key that references other fields, without having to fill it out. The check button was used to allow users to select a 2-way choice such as adding an extra bed or not.

```
self.lb_ids = tk.Listbox(form_frame)
```

Code 5.1 List box

```
item_cb = ttk.Combobox(form_frame, width=28,textvariable=self.room_id)
```

Code 5.2. Combo box

```
tk.Checkbutton(form_frame, variable=self.extra_bed, width=30, text="add",bd=1).grid(row=11, column=1)
```

Code 5.3. Check button

c. Filter function

A filtering function was added to simplify the process of users looking up a specific record. On the customer table, the user can filter records by the last name, search staff table by the last name as well, room type table by the title of room type, room by the room type id, and booking by customer id. By showing all partially matched results, it allows users to get the desired result without taking case sensitivity and minor mistakes into account.

```
def find_by_lastname(self, lastname):
    """
    Filter customer records by customer lastname

    Parameters: customer lastname

    Return: data search result
```

```
    """
    # Print info for debugging
    print("\nFinding customer(s) by lastname ...\n")
    if lastname:
        print(f"lastname: {lastname}")
    else:
        print("no text input")

    # Create a blank dictionary to return the result
    result = {}

    # Using Parameterised Query
    conn = None
    try:
        conn = sqlite3.connect(DATABASE_URI)
        cur = conn.cursor()
        query = "SELECT * FROM customer WHERE lastname LIKE ?;" # Partial match
        # query = "SELECT * FROM customer WHERE lastname = ?;" # Exact match
        param_tuple = (lastname, ) # If a single value, must have a comma at the end!
        cur.execute(query, param  tuple)
            . . .
    return result  # return the result as a dictionary
```

Code 6. Filter function

- # Coding

a. Use doctring comments

Doctring comments are used for every class and method with 4 spaces of indentation to increase the readability and understandability of the source code for future maintenance purposes. Every method and function contain the following three parts: the purpose of the method and function, parameters, and returning values.

```
class CustomerDAO():
    """CustomerDAO class to perform CRUD operations on the customer table in the database"""

    def create(self, data):
        """
        Create/insert a record in a table

        Parameters: data input

        Return: data insertion result
        """
        # Print info for debugging
        print("\nCreating a customer ...\n") #\n means print("\n") a blank line
        print(f"data: {data}")

        result = {}
        . . .
```

Code 7. Doctring comments on Class and Method

b. Using correct Python PEP naming conventions

Since one of the design goals of Python is to be very readable, compliant code will make users take less effort to comprehend the code. Along with the PEP naming conventions, all the variable names follow the lowercase with an underscore and all the class names follow a format of PascalCase; uppercase without underscore.

```
class CustomerDAO()
. . .
inserted_customer_id = cur.lastrowid
. . .
```

Code 8. PEP naming convention

c. Limited length of code/comments

Along with the practice learned from the tutorial, the length of code and comments have been limited to a maximum of 80 characters per line for readability purpose.

d. Parameterized coding

Parameterized style of coding was used to insert a record into an SQL statement to avoid having repeated code with different values and generalize the code.

```
class StaffDAO():
    """StaffDAO class to perform CRUD operations on the staff table in the database"""

    def create(self, data):
                . . .
        result = {}

        # Parameterized Query i.e. question marks as placeholders for  actual values
        conn = None # First initialise the connection to None
        try:
            conn = sqlite3.connect(DATABASE_URI)
            cur = conn.cursor()
            query = "INSERT INTO staff VALUES (?, ?, ?, ?, ?);" # all columns + PK
            param_tuple = (
                None, # staff_id is set to None for database to autoincrement
                data['first_name'],
                data['lastname'],
                data['email'],
                data['title']
            )
            cur.execute(query, param_tuple)
            result['message'] = 'Staff added successfully!'
```

Code 9. Parameterized coding

e. Perform testing and validation

Testing and validation were performed to track and prevent possible exceptions in advance. Testing code of creating a database, methods of DAO classes were written in procedural programming style.

```
def test_is_phone_number(validation):
    """
    A function to test if input follows phone number format of 0000-000-000 or 0000 000 000
```

```
    Parameters: Data to validate.

    Return: None
    """
    print("\n4. Testing is_phone_number()")

    # True
    assert (validation.is_phone_number("0223 999 999"))
    assert (validation.is_phone_number("0456-999-999"))

    # False
    assert (not validation.is_phone_number("(02) 9999 9999"))
    assert (not validation.is_phone_number("0299999999"))
    assert (not validation.is_phone_number("02.9999.9999"))
```

Code 10. Testing and validation code for possible exceptions