Lee En-Yi Hannah
Heng Jeshuan
Yune Thiri Khin
Jheong Rong Yan

Group 2

Overview

The app we proposed to develop is called ‘SQLodge’, which is a platform for accommodation sharing. ‘SQLodge’ will allow property owners to rent out their homes or rooms to travellers. ‘SQLodge’ is an application where users can search for a variety of lodging options including apartments, houses and condominiums. The website offers a simple and intuitive interface for users to book accommodations directly through the site. In order to showcase the maximum utilisation of SQL, we will be primarily focusing on the perspective of property owners. Many of the functions that we will be implementing for property owners can also be used by guests and admin. Here is the link to access our website and Git:

- Website: http://172.25.77.191:5016
- Git: https://github.com/Jheongry/SQLodge

We will organise our SQL queries based on web pages and highlight more complex queries. We will also demonstrate necessary components of HTML, Javascript, and Flask.

Database Creation

The ER diagram is shown in Figure 1, indicating entities, attributes and their relationships.

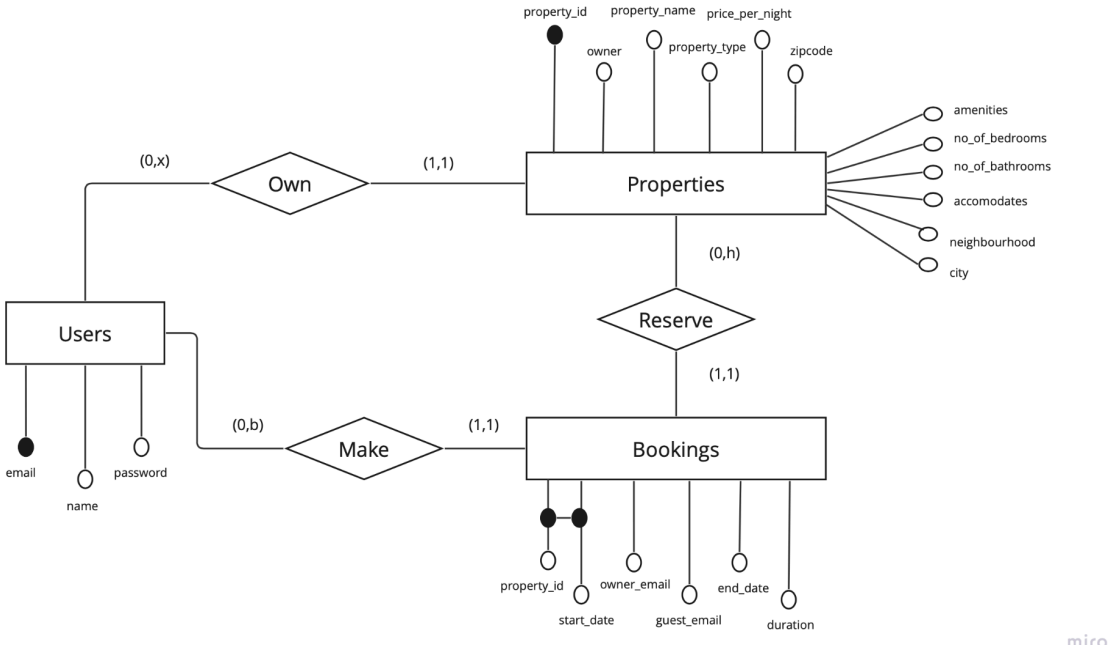

Figure 1: ER Diagram for SQLodge

Based on the ER diagram, we created our database through defining tables, primary keys, foreign keys, and other relevant constraints to establish their relationships.

```
CREATE TABLE IF NOT EXISTS users (
    name VARCHAR(32) NOT NULL,
    email VARCHAR(50) PRIMARY KEY,
    password VARCHAR(16) NOT NULL
);

CREATE TABLE IF NOT EXISTS properties (
    property_id INTEGER PRIMARY KEY,
    owner VARCHAR(50) NOT NULL,
    property_name VARCHAR(100) NOT NULL,
    property_type VARCHAR(32) NOT NULL,
    price_per_night NUMERIC NOT NULL,
    zipcode NUMERIC(5),
    city VARCHAR(32),
    neighbourhood VARCHAR(32),
    square_feet NUMERIC,
    accomodates INTEGER,
    no_of_bathrooms INTEGER,
    no_of_bedrooms INTEGER,
    amenities VARCHAR(1000),

    FOREIGN KEY (owner) REFERENCES users (email) ON UPDATE CASCADE ON DELETE CASCADE
    );

CREATE TABLE IF NOT EXISTS bookings (
    guest_email VARCHAR(50),
    owner_email VARCHAR(50),
    property_id INTEGER,
    start_date DATE NOT NULL,
    end_date DATE NOT NULL,
    duration NUMERIC,

    FOREIGN KEY (guest_email) REFERENCES users (email) ON UPDATE CASCADE ON DELETE
CASCADE,
    FOREIGN KEY (owner_email) REFERENCES users (email) ON UPDATE CASCADE ON DELETE
CASCADE,
    FOREIGN KEY (property_id) REFERENCES properties (property_id) ON UPDATE CASCADE ON
DELETE CASCADE,

    PRIMARY KEY (property_id, start_date),
    CHECK (guest_email != owner_email),
    CHECK (start_date < end_date)
);
```

The tables were populated using data found on Kaggle and Mockaroo and which is stored under the formatted_data folder in csv files. The following code is an example of inserting data into the 'properties' table. The same technique was applied to other values and tables.

```
INSERT INTO properties VALUES (25138355,'iboath6d@yandex.ru','Private Bedroom in Ocean
Beach Home','Apartment',50.0,92107,'San Diego','Ocean Beach',NULL,1,1.0,1.0,
'TV,Wifi,Kitchen,Washer,Dryer,Smoke detector,Carbon monoxide
detector,Essentials,Hangers,Iron,Laptop friendly workspace' );
```

App Structure

App.py defines a Flask application and configures it to use Flask-Login and Flask-Admin extensions. It registers blueprints for authentication and views. A custom view named MyView is also constructed to handle the root URL of the application and render an index.html template.

In 'models.py', necessary modules are imported to set up a SQLAlchemy engine for connecting to a PostgreSQL database. Flask-Login's UserMixin class is imported for user authentication. The Users, Properties, and Bookings classes are created by reflecting on the tables in the PostgreSQL database schema. For instance, Users class is later used in auth.py where more complex flask_login functions require a class as an argument.

Main.py starts the Flask application by running it with app.run() method, which binds the application to the IP address 0.0.0.0 and the port number 5019.

App Functions

The following part demonstrates the main functions of the app and different Python files, HTML files and flask forms are involved for different utilities. While testing, we recommend that you log in with the following email and password to try out these functionalities:

Email: alevett65@wufoo.com Password: NOOlabAC

1. Login Page

   Upon entering the website, users will be prompted to sign in with their email and password. The input is obtained using LoginForm that is classed under Flaskform, with HTML creating the interface. Other forms also adopt the same structure but with different inputs such as BookingForm and others.

   Our login and logout routes are created with the help of flask's Blueprint class and the flask_login library. For both, sqlalchemy queries were used in place of SQL queries as they return a query

object, which was needed for further authentication functions as shown below in this excerpt of our login function in 'auth.py':

```
class LoginForm(FlaskForm):
   email = StringField('Email',
                       validators=[DataRequired(), Email()])
   password = PasswordField('Password', validators=[DataRequired()])
   remember = BooleanField('Remember Me')
   submit = SubmitField('Login')
```

The below code checks if a user with the given email and password exists in the database. If it does, the user logs in successfully, otherwise, they are prompted to create a new account.

```
user =
session.query(Users).filter_by(email=f'{form.email.data}',password=f'{form.pass
word.data}').first()
logging.warning(user)
if user:
       flash(f'Login Successful for {form.email.data}', 'success')
       login_user(user, remember=True)
       return redirect(url_for('views.home'))
else:
```

The 'register' function inserts the provided username, email, and password into the user database. The email is used as the primary key, hence if the provided email already exists in the table, the user is informed to use a different email via a flash message.

```
@auth.route("/register", methods=['GET', 'POST'])
def register():
   form = RegistrationForm()
   user = None
   if request.method == 'POST':
       if form.validate_on_submit():
           statement = sqlalchemy.text(f"SELECT * FROM users WHERE
email='{form.email.data}' AND password='{form.password.data}';")
           user = session.execute(statement).first()
           if user:
               flash('Email already exists.', category='error')
               return redirect(url_for('auth.register'))

           else:
               try:
                   statement = sqlalchemy.text(f"INSERT INTO users VALUES
('{form.username.data}', '{form.email.data}', '{form.password.data}');")
                   db.execute(statement)
                   db.commit()
```

2. Listings page

The 'listings' page shows all properties and their details. Users are able to filter and sort through the listings depending on the criteria they are looking for. In order to do this, user input is first obtained through the FilterForm that is classed under FlaskForm, with the listings.html file creating the interface, as shown below:

```
class FilterForm(FlaskForm):
    property_type = SelectField('Property Type',
                                choices=[('None', 'All'), ('Apartment',
'Apartment'), ('House', 'House'), ('Condominium', 'Condominium'),
('Others','Others')])
    min_price = IntegerField('Minimum Price Per Night')
    max_price = IntegerField('Maximum Price Per Night')
    neighbourhood = SelectField('Neighbourhood',
                                choices = [('All', 'All'), ('Pacific
Beach','Pacific Beach'), ('Tierrasanta','Tierrasanta'), ('Rancho
Penasquitos','Rancho Penasquitos'), ('Point Loma Heights','Point Loma
Heights'), ('Mission Beach','Mission Beach'), ('East Village','East Village'),
('Park West','Park West'), ('Little Italy','Little Italy'), ('Cherokee
Point','Cherokee Point'), ('North Park','North Park'), ('Bay Park','Bay Park'),
('Sherman Heights','Sherman Heights')])

    except_neighbourhood = SelectField('Except Neighbourhood',
                                    choices=[('None', 'None'), ('Pacific
Beach','Pacific Beach'), ('Tierrasanta','Tierrasanta'),('Rancho
Penasquitos','Rancho Penasquitos'), ('Point Loma Heights','Point Loma
Heights'),('Mission Beach','Mission Beach'), ('East Village','East Village'),
('Park West','Park West'), ('Little Italy','Little Italy'), ('Cherokee
Point','Cherokee Point'), ('North Park','North Park'), ('Bay Park','Bay Park'),
('Sherman Heights','Sherman Heights')])
    price_sort_by = RadioField('Sort by Price Per Night', choices=[('ASC', 'Low
to High'), ('DESC', 'High to Low')]
                           , default = ('ASC', 'Low to High'))
    bookings_sort_by = RadioField('Sort by Bookings',
                                  choices=[('ASC', 'Least Booked to Most
Booked'), ('DESC', 'Most Booked to Least Booked')],
                                  default=('ASC', 'Least Booked to Most
Booked'))
    submit = SubmitField('Filter')
```

Due to the nature of this website being a demo, the neighbourhoods are limited to the 12 choices shown above.

User input is then used to formulate the SQL statements to query the database such that it returns the filtered listings based on the user's criteria. Below is a snippet of the function to filter the listings:

```
        if form.validate_on_submit():
            property_type = form.property_type.data
            min_price = form.min_price.data
            max_price = form.max_price.data
            neighbourhood = form.neighbourhood.data
            except_neighbourhood = form.except_neighbourhood.data
            price_sort_by = form.price_sort_by.data
            bookings_sort_by = form.bookings_sort_by.data
            statement = "SELECT p.* FROM properties p, bookings b WHERE
b.property_id = p.property_id AND b.owner_email = p.owner"
            if property_type:
                if (property_type == 'Others'):
                    statement += f" AND p.property_type NOT IN ('House',
'Apartment', 'Condominium')"
                elif (property_type != 'None'):
                    statement += f" AND p.property_type ='{property_type}'"
            if min_price:
                statement += f" AND p.price_per_night >= {min_price}"
            if max_price:
```

```
                statement += f" AND p.price_per_night <= {max_price}"
            if neighbourhood:
                if (neighbourhood != 'All'):
                    statement += f" AND p.neighbourhood ='{neighbourhood}'"
            if except_neighbourhood:
                if (except_neighbourhood != 'None'):
                    statement += f" AND p.neighbourhood
!='{except_neighbourhood}'"
            if bookings_sort_by:
                statement += f" GROUP BY p.property_id ORDER BY COUNT(*)
{bookings_sort_by}"
            if price_sort_by:
                statement += f" , price_per_night {price_sort_by}"

            statement = sqlalchemy.text(statement)
            listings = db.execute(statement)
            listings = generate_table_return_result(listings)
            listings = json.loads(listings)
            return render_template('listings.html', title='Listings',
form=form, table_data=listings, user=current_user)
        return render_template('listings.html', title='Listings', form=form,
table_data=data, user=current_user)
```

If the user wishes to book a listing, they can click the property id of the desired listing, which will bring them to the booking page.

3. Booking page

Users are required to provide their email, check-in check-out date to make a booking. The relevant property ID and owner's email will also be shown but no input is required.

It is necessary for the guest's email to match the account email and this is ensured through the implementation of the following code.

```
    if form.validate_on_submit():
        try:
            statement = sqlalchemy.text(f"SELECT * FROM users u WHERE
u.email='{form.guest_email.data}';")
            guest = db.execute(statement)
            db.commit()
        except Exception as e:
            db.rollback()
            flash(str(e), 'error')
```

The code ensures the start date is not earlier than today's date and that the end date is not earlier than the start date.

```
 try:
            start_date= form.start_date.data
            end_date = form.end_date.data
            # Check that the start date is not earlier than today's date
            if start_date < datetime.now().date():
                flash('Start date cannot be earlier than today.',
category='error')
                return
redirect(f"/booking?property_id={form.property_id.data}")
```

```
            # Check that the end date is not earlier than the start date
            if end_date < start_date:
                flash('End date cannot be earlier than start date.',
category='error')
                return
redirect(f"/booking?property_id={form.property_id.data}")
```

Lastly, another SQL query is constructed to search for any bookings on the same property that overlap with the requested start and end dates.

```
 statement = sqlalchemy.text(f"SELECT * FROM bookings WHERE property_id
='{form.property_id.data}' AND (('{form.start_date.data}' BETWEEN start_date
AND end_date) OR ('{form.end_date.data}' BETWEEN start_date AND end_date));")
            conflict_booking = db.execute(statement).fetchall()
            if conflict_booking:
                flash('Booking dates conflict with another booking on this
property.', category='error')
                return
redirect(f"/booking?property_id={form.property_id.data}")
```

If no conflict is found, the booking is successful and will be added into the record.

```
 try:
            statement = sqlalchemy.text(f"INSERT INTO bookings VALUES
('{form.guest_email.data}', '{form.owner_email.data}',
'{form.property_id.data}', '{form.start_date.data}',  '{form.end_date.data}',
{days});")
            db.execute(statement)
            db.commit()
            flash('Your booking has been confirmed!', 'success')
            return redirect(f"/booking?property_id={form.property_id.data}")
        except sqlalchemy.exc.SQLAlchemyError as e:
            db.rollback()
            flash('Your email must match your account email.',
category='error')
```

4. My Listings page

Only properties which the user owns will be shown on this page. The delete function, which uses the SQL DELETE statement, is available if they wish to delete a particular property. As only properties owned by user will be shown, they can only delete their own listings.

```
@views.route('/delete_entry', methods=['POST'])
def delete_entry():
   id = request.args.get('id')
   statement = sqlalchemy.text(f"DELETE FROM properties WHERE property_id =
{id};")
   db.execute(statement)
   db.commit()
   flash('Property deleted successfully!', 'success')
   return redirect(url_for('views.get_my_listings'))
```

Upon clicking on the property id, user will enter the update listings page.

5. Update Listings page

While in the /mylistings page, users can click on the property_id of their listing to access its /updatemylistings page. As the property_id is the primary key, we do not allow users to update the property_id, though it will be displayed within the form. As the /updatemylistings page can only be accessed through the /mylistings page mentioned in part 4, user will only be allowed to update the property which belongs to them. Trying to access this page via the url directly will result in an error. By providing the updated details, such as the number of people it can accommodate or the amenities available, they will be able to update their listing.

```
@views.route('/updatemylistings', methods=['GET', 'POST'])
@login_required
def update_listings():
    form = UpdatePropertiesForm()

    if request.method == 'POST' and form.validate_on_submit():
        try:
            statement = sqlalchemy.text(f"UPDATE properties SET price_per_night
= '{form.price_per_night.data}', accomodates  = '{form.accommodates.data}',
amenities = '{form.amenities.data}' WHERE property_id =
'{form.property_id.data}';")
            db.execute(statement)
            db.commit()
            flash('Your listing has been updated!', 'success')
            return redirect(url_for('views.get_my_listings'))
```

While not shown in the above snippet, the form is also pre-filled similar to the booking page in part 3.

6. Profile page

Users can view their name and email on this page. They can update their name and email by providing a new name and email. If the email/name already exists in the database, there will be an error and they will be redirected back to /profile and prompted to provide a new one as the user is already taken.

```
@views.route("/profile", methods=['GET', 'POST'])
@login_required
def profile():
    form = UpdateAccountForm()
    if form.validate_on_submit():
        if form.email.data != current_user.email:
            statement = sqlalchemy.text(f"SELECT * FROM users u WHERE
u.email='{form.email.data}';")
            guest = db.execute(statement).fetchone()
            db.commit()
            logging.warning(guest)
            if guest:
                flash('That email is taken. Please choose a different one.',
category='error')
                return redirect(url_for('views.profile'))
        if form.name.data != current_user.name:
            statement = sqlalchemy.text(f"SELECT * FROM users u WHERE
u.name='{form.name.data}';")
            guest = db.execute(statement).fetchone()
```

```
            db.commit()
            if guest:
                flash('That name is taken. Please choose a different one.',
category='error')
                return redirect(url_for('views.profile'))
```

If both email and name are new, the new email and name will be updated in the database. Users will be brought back to the login page and will be prompted to log in again with their new email.

```
        # if both new name and email are valid
        try:
            statement = sqlalchemy.text(f"UPDATE users SET email =
'{form.email.data}', name = '{form.name.data}' WHERE email =
'{current_user.email}';")
            db.execute(statement)
            db.commit()
            flash('Your account has been updated! Please log in again.',
'success')
            return redirect(url_for('auth.logout'))
```

7. My Statistics page

After keying in a user email, users are able to view some popular statistics linked to the user email, as shown in Figure 2.

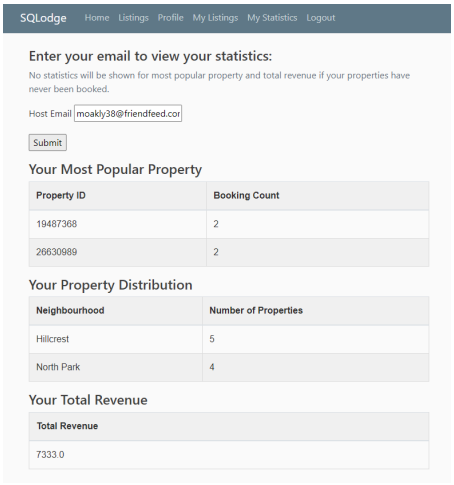

Figure 2. Statistics Page

These statistics make use of aggregate and nested queries to provide greater insights into a user's properties, bookings and revenue.

```
@views.route("/")
@views.route("/statistics", methods=["GET", "POST"])
@login_required
def statistics():
    form = StatsForm()
    data, data1, data2 = None, None, None
    if form.validate_on_submit():
        try:
            statement = sqlalchemy.text(f"SELECT property_id, COUNT(*) as count
FROM bookings WHERE owner_email = '{form.email.data}' GROUP BY property_id
```

```
HAVING COUNT(*) >= ALL (SELECT COUNT(*) FROM bookings WHERE owner_email =
'{form.email.data}' GROUP BY property_id);")
            res = db.execute(statement)
            data = generate_table_return_result(res)
            data = json.loads(data)
            logging.warning(data)

            statement = sqlalchemy.text(f"SELECT neighbourhood, COUNT(*) as
count FROM properties WHERE owner = '{form.email.data}' GROUP BY
neighbourhood;")
            res = db.execute(statement)
            data1 = generate_table_return_result(res)
            data1 = json.loads(data1)

            statement = sqlalchemy.text(f"SELECT
SUM(p.price_per_night*duration) FROM properties p, bookings b WHERE
b.owner_email = p.owner AND p.property_id = b.property_id AND p.owner =
'{form.email.data}';")
            res = db.execute(statement)
            data2 = generate_table_return_result(res)
            data2 = json.loads(data2)

        except Exception as e:
            db.rollback()
            return Response(str(e), 403)
    return render_template('statistics.html', title='Statistics', form=form,
stats_data=data, stats_data1=data1, stats_data2=data2, user=current_user)
```

8. Admin Interface

   Our admin interface can be accessed through the /admin path or through the link on our /home page. The admin interface is created using the flask_admin library and allows administrators to view our three tables, Users, Properties and Bookings, and delete, update and create entries within these tables. As SQL queries needed for such administrator functions have already been demonstrated in our other app functions, we have chosen to focus on the implementation of those functions rather than repeat similar SQL queries for the admin interface.