데이터베이스 ESQL 과제

- 웹언어 html –

2315028 인공지능공학부 김성현

1. Progress Report

2024.11.09

PM10:00-1:00 주제설정 및 데이터 생성

2024.11.15

PM2:00-5:00 웹언어로 함수를 구현하고, 프로그램 실행

2024.11.17~18

고급언어와 연동

2024. 11. 20~23

코드 수정 (오류 수정)

2024. 11. 23

PM 9-11:00 보고서 작성

2. 주제

눈송이 은행원 프로그램

: 눈송이 은행에서 일하는 직원이 사용하는 프로그램을 구현해보았습니다.
고객의 계좌 관리부터 지점관리까지 모든 것이 가능한 프로그램을 구현하였습니다.

3. Html & javascript

html

```
0.  <!DOCTYPE html>
1.  <html lang="ko">
2.  <head>
3.      <meta charset="UTF-8">
4.      <meta name="viewport" content="width=device-width, initial-scale=1.0">
5.      <title>눈송이 은행</title>
6.      <style>
7.          body {
8.              font-family: Arial, sans-serif;
9.              text-align: center;
10.             background-color: #f4f8fb;
11.         }
12.         .container {
```

```
            margin: 50px auto;
            max-width: 600px;
            background: #ffffff;
            padding: 20px;
            border-radius: 10px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }
        button {
            margin: 10px;
            padding: 10px 20px;
            font-size: 1rem;
            border: none;
            border-radius: 5px;
            background-color: #007bff;
            color: #ffffff;
            cursor: pointer;
        }
        button:hover {
            background-color: #0056b3;
        }
        .result {
            margin-top: 20px;
            padding: 15px;
            border: 1px solid #ddd;
            border-radius: 5px;
            background: #f9f9f9;
            text-align: left;
        }
    </style>
</head>
```

1. **헤드 섹션**: 페이지의 메타 정보와 스타일이 정의되어 있습니다.

•<meta charset="UTF-8">: 페이지의 문자 인코딩을 UTF-8 로 설정하여 한글을 포함한 다양한 문자를 문제 없이 표시할 수 있습니다.
•<meta name="viewport" content="width=device-width, initial-scale=1.0">: 이 설정은 페이지를 모바일 장치에서도 잘 보이도록 만들어 줍니다. width=device-width 는 화면 너비에 맞게 페이지를 조정하고, initial-scale=1.0 은 페이지의 초기 확대 비율을 1 로 설정합니다.
•<title>눈송이 은행</title>: 브라우저 탭에 표시될 페이지 제목을 "눈송이 은행"으로 설정합니다.
•<style>: 페이지의 스타일을 설정하는 CSS 코드가 포함되어 있습니다. 여기서는 배경색, 버튼 스타일, 결과 출력 영역 등의 스타일이 설정되어 있습니다.

```
<body>
    <div class="container">
        <h1>눈송이 은행</h1>
        <div id="menu">
```

```
            <button onclick="fetchUserInfo()">사용자 정보 조회</button>
            <button onclick="fetchAccountHistory()">거래내역 조회</button>
            <button onclick="updateAccount()">입출금</button>
            <button onclick="createUser()">신규 사용자 생성</button>
            <button onclick="fetchBranchInfo()">은행지점 조회</button>
            <button onclick="exit()">종료</button>
        </div>
        <div id="result" class="result"></div>
    </div>
```

2. **본문 섹션** :

<h1>눈송이 은행</h1>: 페이지 상단에 "눈송이 은행"이라는 제목을 표시합니다. 이는 사이트의 이름을 나타냅니다.

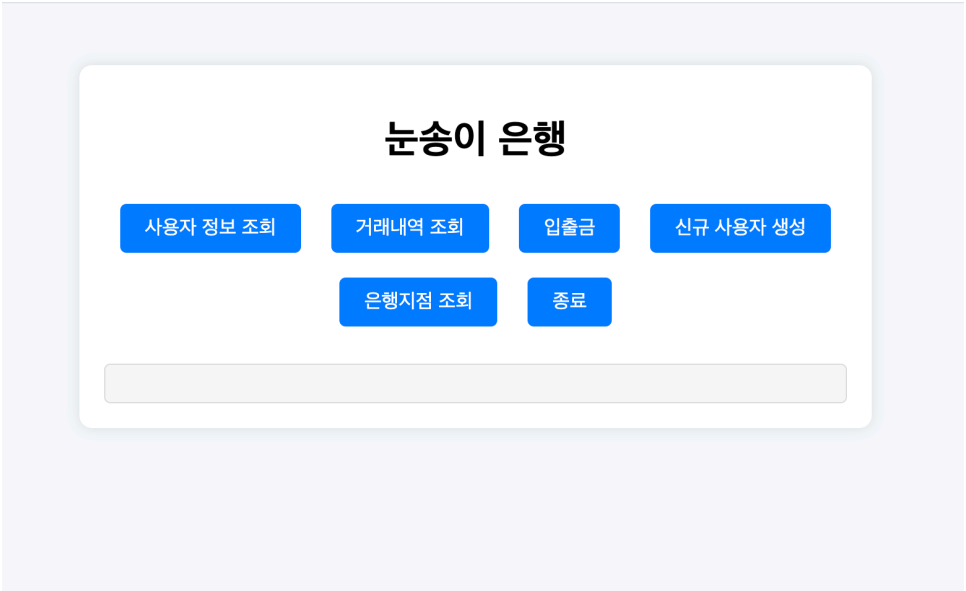
: 여러 개의 버튼이 포함된 div 요소로, 사용자가 클릭할 수 있는 여러 기능을 제공합니다.

버튼들: 각 버튼은 사용자가 선택할 수 있는 은행 관련 작업을 나타냅니다.

- 사용자 정보 조회: 사용자의 정보를 조회하는 기능을 제공합니다.
- 거래내역 조회: 특정 계좌의 거래내역을 조회하는 기능입니다.
- 입출금: 사용자가 입금 또는 출금할 수 있는 기능을 제공합니다.
- 신규 사용자 생성: 새로운 사용자를 추가하는 기능입니다.
- 은행지점 조회: 특정 지점에 대한 정보를 조회하는 기능입니다.
- 종료: 애플리케이션을 종료하는 버튼입니다.

: 이 div 는 API 호출 결과를 표시하는 영역입니다. 각 API 호출이 끝나면 여기서 결과를 확인할 수 있습니다.

출력



Javascript

```
<script>
        const API_BASE_URL = 'http://127.0.0.1:5000';
```

1. **API_URL 설정**:

const API_BASE_URL = 'http://127.0.0.1:5000';로 로컬 서버에 있는 API와 통신합니다.

```
async function fetchUserInfo() {
            const userId = prompt("사용자 ID를 입력하세요:");
            if (!userId) return alert("ID를 입력하세요.");
            try {
                const response = await fetch(`${API_BASE_URL}/user/${userId}`);
                if (!response.ok) throw new Error("사용자 정보를 가져오는 데 실패했습니다.");
                const data = await response.json();
                displayResult(data, "사용자 정보");
            } catch (error) {
                alert(error.message);
            }
        }
```

2. **fetchUserInfo 함수**:

사용자 ID를 입력받고 해당 ID로 사용자의 정보를 조회합니다.
prompt()를 사용하여 사용자에게 ID를 입력받고, 이를 바탕으로 API를 호출합니다.
호출이 성공하면 displayResult 함수를 호출하여 데이터를 화면에 표시합니다. 실패 시 오류 메시지가 나타납니다.

- **오류 처리**:

입력값 검증: 사용자가 아무것도 입력하지 않았을 경우 alert를 통해 경고를 표시.
HTTP 응답 오류: response.ok가 false일 경우, throw를 통해 오류를 발생시켜 catch 블록으로 전달.
네트워크 오류: API 호출 중 네트워크 오류가 발생하면 catch 블록에서 처리하며 오류 메시지를 표시.

```
        async function fetchAccountHistory() {
            const accountNumber = prompt("계좌번호를 입력하세요:");
            if (!accountNumber) return alert("계좌번호를 입력하세요.");
            try {
                const response = await fetch(`${API_BASE_URL}/account/history/${accountNumber}/transactions`);
                if (!response.ok) throw new Error("거래내역을 가져오는 데 실패했습니다.");
                const data = await response.json();
                displayResult(data, "거래내역");
            } catch (error) {
                alert(error.message);
```

```
        }
    }
```

3. **fetchAccountHistory 함수**:

계좌번호를 입력받고, 해당 계좌의 거래 내역을 조회하는 함수입니다. fetch()로 API 요청을 보내고, 응답 데이터를 받아서 displayResult 함수로 출력합니다.

- **오류 처리**:

입력값 검증: 계좌번호가 입력되지 않은 경우 경고 메시지를 표시.
HTTP 응답 오류: API 응답이 실패하면 사용자에게 실패 이유를 알림.
네트워크 오류: 네트워크 연결 문제 발생 시 오류 메시지를 출력.

```
async function updateAccount() {
        const accountNumber = prompt("계좌번호를 입력하세요 :");
        const transactionType = prompt("입금/출금 중 입력하세요 : ");
        const amount = prompt(`${transactionType}할 금액을 입력하세요 :`);

        const data = {
            accountNumber: accountNumber,
            amount: amount,
            transactionType: transactionType
        };

        try {
            const response = await fetch(`${API_BASE_URL}/account/update`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify(data)
            });

            const result = await response.json();
            if (response.ok) {
                alert(result.message);
            } else {
                alert(result.error);
            }
        } catch (error) {
            alert(error.message);
        }
    }
```

4. **updateAccount 함수**:

사용자가 계좌번호, 입금/출금 종류, 금액을 입력한 후, 이를 서버에 전달하여 입출금을 처리하는 함수입니다.

API 에 POST 방식으로 데이터를 전송하고, 응답에 따라 결과를 alert()로 사용자에게 알립니다.

오류 처리:

HTTP 응답 상태 검사: API 호출이 성공했는지 여부를 확인하고, 실패 시 서버에서 전달한 오류 메시지를 표시.

네트워크 오류: 서버가 응답하지 않거나 연결 문제가 발생했을 경우 catch 블록에서 처리.

```
async function createUser() {
        const userData = {
            user: [
                prompt("사용자 ID:"),
                prompt("이름:"),
                prompt("성별 (M/F):"),
                prompt("연락처 (010-0000-0000):"),
                prompt("주민번호 (900101-1234567):"),
                prompt("지점 번호:")
            ],
            account: [
                prompt("계좌번호:"),
                prompt("사용자 ID:"),
                prompt("계좌유형 (저축/당좌/정기예금):"),
                prompt("초기 잔액:"),
                prompt("지점 번호:")
            ]
        };
        try {
            const response = await fetch(`${API_BASE_URL}/user`, {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(userData)
            });
            if (!response.ok) throw new Error("사용자 및 계좌 생성에 실패했습니다.");
            const data = await response.json();
            alert(data.message);
        } catch (error) {
            alert(error.message);
        }
    }
```

5. **createUser 함수**:

새 사용자를 생성하는 기능으로, 여러 가지 사용자 정보와 계좌 정보를 prompt()로 입력받아 서버에 POST 방식으로 전송합니다.

서버 응답에 따라 생성 성공 또는 오류 메시지를 alert()로 사용자에게 표시합니다.

오류 처리 :

HTTP 응답 상태 검사: API 호출이 실패하면 오류 메시지를 표시.

네트워크 오류: 네트워크 문제 발생 시 사용자에게 오류 메시지를 전달.

```
async function fetchBranchInfo() {
        const branchId = prompt("지점 ID 를 입력하세요:");
        if (!branchId) return alert("지점 ID 를 입력하세요.");
        try {
            const response = await fetch(`${API_BASE_URL}/branch/${branchId}`);
            if (!response.ok) throw new Error("지점 정보를 가져오는 데 실패했습니다.");
            const data = await response.json();
            displayResult(data, "지점 정보");
        } catch (error) {
            alert(error.message);
        }
    }
```

6. **fetchBranchInfo 함수**:

지점 ID를 입력받고, 해당 지점의 정보를 조회하는 함수입니다.

API에서 데이터를 가져와 displayResult 함수를 사용해 화면에 표시합니다.

오류 처리:

입력값 검증: 지점 ID를 입력하지 않았을 경우 경고 메시지를 출력.

HTTP 응답 상태 검사: API 응답이 실패한 경우 사용자에게 실패 메시지 제공.

네트워크 오류: 네트워크 연결 문제 처리.

```
function displayResult(data, title) {
        const resultDiv = document.getElementById('result');
        resultDiv.innerHTML = `<h2>${title}</h2><pre>${JSON.stringify(data, null, 2)}</pre>`;
    }
```

7. **displayResult 함수**:

API에서 받은 데이터를 화면에 예쁘게 표시하는 함수입니다. JSON.stringify(data, null, 2)로 데이터를 들여쓰기하여 읽기 쉽게 포맷팅합니다.

결과는 pre 태그로 감싸져 출력되며, 결과 제목(title)도 함께 표시됩니다.

```
async function exit() {
    if (confirm("프로그램을 종료하시겠습니까?")) {
        window.close();
        alert("눈송이 은행이 종료되었습니다! *브라우저창을 닫아주세요*")
    }
```

```
    }
```

8. **exit 함수**:

종료 버튼을 클릭했을 때 실행됩니다. 사용자에게 종료 여부를 확인하는 confirm() 창을 띄운 후, 확인을 클릭하면 window.close()로 브라우저 창을 닫습니다.

종료 후에는 "눈송이 은행이 종료되었습니다!"라는 메시지를 alert()로 표시합니다.

오류 처리 :
window.close()는 일부 브라우저에서 동작하지 않을 수 있으므로 종료 메시지를 추가로 표시하여 안내.

4. Flask

```
from flask import Flask, jsonify, request
from datetime import datetime
import mysql.connector
from flask_cors import CORS
import logging
from decimal import Decimal

app = Flask(__name__)
CORS(app)

# 로깅 설정
logging.basicConfig(level=logging.DEBUG, format="%(asctime)s [%(levelname)s] %(message)s")
```

• CORS: 브라우저의 **Cross-Origin Resource Sharing** 문제를 해결하여 다른 도메인에서 API 요청 허용.

• logging: 서버에서 발생하는 주요 동작이나 오류를 기록.
로깅 설정은 DEBUG 레벨로 되어 있어 모든 세부 정보가 로그에 기록됩니다.

```
def connect_db():
    return mysql.connector.connect(
        host="localhost",
        user="root",
        password="khyun0303",
        database="ESQL"
    )
```

1. connect_db() 함수

- **역할**: MySQL 데이터베이스 연결을 생성하여 쿼리를 실행할 수 있는 환경을 제공합니다.
- **오류 처리**: 연결 실패 시 MySQL 라이브러리가 자동으로 예외를 발생시키며, 연결이 제대로 닫히지 않으면 with 문이 안전하게 리소스를 해제합니다.

```
def execute_select_query(query, params):
```

```
    with connect_db() as db:
        with db.cursor() as cursor:
            cursor.execute(query, params)
            return cursor.fetchall()

def execute_commit_query(query, params):
    with connect_db() as db:
        with db.cursor() as cursor:
            cursor.execute(query, params)
            db.commit()
```

2. 공통 쿼리실행 함수:

- **execute_select_query** : SELECT 쿼리를 실행하여 결과를 반환합니다.
- **execute_commit_query** : NSERT/UPDATE 쿼리를 실행하고 변경 사항을 커밋합니다.

두 함수 모두 with **문**을 사용하여 연결과 커서를 안전하게 관리하고, 데이터베이스 리소스를 자동으로 반환합니다. 이로써 **리소스 누수**를 방지합니다.

```
@app.route('/user/<int:user_id>', methods=['GET'])
def get_user_info(user_id):
    query = """
    SELECT 사용자.이름, 사용자.성별, 사용자.연락처, 계좌.계좌번호
    FROM 사용자
    JOIN 계좌 ON 사용자.사용자_ID = 계좌.사용자_ID
    WHERE 사용자.사용자_ID = %s;
    """
    result = execute_select_query(query, (user_id,))
    data = [{"이름": r[0], "성별": r[1], "연락처": r[2], "계좌번호": r[3]} for r in result]
    return jsonify(data)
```

3. 사용자정보조회 API

- **역할**: 사용자 ID 로 해당 사용자와 계좌 정보를 조회하여 반환합니다.
- **오류 처리**: 이 API 는 오류를 명시적으로 처리하지 않으므로, 잘못된 user_id 로 인한 빈 결과를 클라이언트가 직접 확인해야 합니다.

```
@app.route('/account/history/<int:account_number>/transactions', methods=['GET'])
def get_account_history(account_number):
    query = """
    SELECT 사용자.이름, 거래내역.거래일시, 거래내역.거래유형, 거래내역.거래금액, 지점.위치
    FROM 거래내역
    JOIN 계좌 ON 거래내역.계좌번호 = 계좌.계좌번호
    JOIN 사용자 ON 계좌.사용자_ID = 사용자.사용자_ID
    JOIN 지점 ON 사용자.지점_ID = 지점.지점_ID
    WHERE 계좌.계좌번호 = %s;
    """
    result = execute_select_query(query, (account_number,))
    data = [
        {
            "이름": r[0],
```

```
            "거래일시": r[1].strftime("%Y-%m-%d %H:%M:%S"),
            "거래유형": r[2],
            "거래금액": r[3],
            "지점위치": r[4]
        } for r in result
    ]
    return jsonify(data)
```

4. 거래내역조회 API

•**역할**: 특정 계좌의 거래내역(거래 유형, 금액, 지점 정보 등)을 조회합니다.

• **오류 처리**: 잘못된 account_number 로 인해 데이터가 없더라도 에러를 반환하지 않으며 빈 목록을 반환. 거래 날짜(거래일시)를 문자열로 변환(strftime)하여 JSON 형식과 호환되도록 처리합니다.

```
@app.route('/account/history/<int:account_number>/balance', methods=['GET'])
def get_account_balance(account_number):
    query = "SELECT 잔액 FROM 계좌 WHERE 계좌번호 = %s;"
    result = execute_select_query(query, (account_number,))
    if result:
        return jsonify({"잔액": result[0][0]})
    else:
        return jsonify({"error": "계좌를 찾을 수 없습니다."}), 404
```

5. 계좌잔액조회 API

• **역할** : 계좌번호를 입력받아 해당 계좌의 잔액을 반환합니다.

• **오류 처리** : 계좌가 없는 경우 **HTTP 404 상태 코드**와 함께 에러 메시지 반환. 데이터가 없을 경우와 정상 응답의 경우를 명확히 구분합니다.

```
@app.route('/account/update', methods=['POST'])
def update_account():
    data = request.json
    account_number = data.get('accountNumber')
    amount = Decimal(data.get('amount'))
    transaction_type = data.get('transactionType')
    balance_query = "SELECT 잔액 FROM 계좌 WHERE 계좌번호 = %s;"
    result = execute_select_query(balance_query, (account_number,))

    if not result:
        return jsonify({"error": "계좌를 찾을 수 없습니다."}), 404
    current_balance = Decimal(result[0][0])
    if transaction_type == "출금" and current_balance < amount:
        return jsonify({"error": "잔액이 부족합니다."}), 400

    # 잔액 업데이트
    if transaction_type == "입금":
        update_query = "UPDATE 계좌 SET 잔액 = 잔액 + %s WHERE 계좌번호 = %s;"
    elif transaction_type == "출금":
        update_query = "UPDATE 계좌 SET 잔액 = 잔액 - %s WHERE 계좌번호 = %s;"
    else:
        return jsonify({"error": "잘못된 거래 유형입니다."}), 400
    execute_commit_query(update_query, (amount, account_number))
```

```
    # 거래내역 기록
    transaction_id_query = "SELECT COALESCE(MAX(거래_ID), 2000) + 1 FROM 거래내역;"
    transaction_id = execute_select_query(transaction_id_query, ())[0][0]
    insert_query = """
    INSERT INTO 거래내역 (거래_ID, 계좌번호, 거래일시, 거래유형, 거래금액)
    VALUES (%s, %s, %s, %s, %s);
    """
    execute_commit_query(insert_query, (transaction_id, account_number, datetime.utcnow(),
transaction_type, amount))
    return jsonify({
        "message": f"{transaction_type} 완료: {amount}원이 계좌에 반영되었습니다.",
        "updated_balance": current_balance + (amount if transaction_type == "입금" else -amount)
    })
```

6. 계좌잔액 업데이트 API

- **역할** : 계좌에 입금 또는 출금 작업을 수행하고 잔액을 업데이트합니다.

거래내역 기록을 위해 새 거래 ID 생성. 트랜잭션 완료 후 성공 메시지와 갱신된 잔액 반환합니다.

- **오류 처리** :

계좌가 존재하지 않으면 **404 에러** 반환. 출금 금액이 현재 잔액보다 크면 **400 에러** 반환합니다.
지원하지 않는 거래 유형(입금/출금이 아닌 경우) 입력 시 **400 에러** 반환합니다.

```
@app.route('/user', methods=['POST'])
def create_user_account():
    data = request.json
    user_data = data["user"]
    account_data = data["account"]
    try:
        user_query = """
        INSERT INTO 사용자 (사용자_ID, 이름, 성별, 연락처, 주민번호, 지점_ID)
        VALUES (%s, %s, %s, %s, %s, %s);
        """
        account_query = """
        INSERT INTO 계좌 (계좌번호, 사용자_ID, 계좌유형, 잔액, 지점_ID)
        VALUES (%s, %s, %s, %s, %s);
        """
        execute_commit_query(user_query, user_data)
        execute_commit_query(account_query, account_data)
        return jsonify({"message": "사용자 및 계좌 생성 성공"})
    except Exception as e:
        logging.error(f"사용자 및 계좌 생성 실패: {str(e)}")
        return jsonify({"error": "사용자 및 계좌 생성 실패"}), 500
```

7. 신규사용자 및 계좌 생성 API

- **역할** : 사용자 정보를 등록하고, 새 계좌를 생성합니다.

- **오류 처리**: 데이터베이스 작업 중 예외가 발생할 경우, **500 에러**를 반환하며 실패 로그를 기록합니다. 입력 데이터가 잘못되거나 누락된 경우도 로그에 포함합니다.

```
# 지점별 사용자 조회 API
@app.route('/branch/<int:branch_id>', methods=['GET'])
```

```python
def get_branch_accounts(branch_id):
    query = """
    SELECT 지점.위치, 지점.지점장_이름, 지점.직원수, COUNT(사용자.사용자_ID) AS 사용자_수
    FROM 지점
    JOIN 사용자 ON 사용자.지점_ID = 지점.지점_ID
    WHERE 지점.지점_ID = %s
    """
    result = execute_select_query(query, (branch_id,))
    data = [
        {
            "위치": r[0],
            "지점장_이름": r[1],
            "직원수": r[2],
            "사용자_수": r[3]
        } for r in result
    ]
    return jsonify(data)
```

8. 지점별 사용자 조회 API

- **역할**: 특정 지점의 위치, 지점장 이름, 직원 수, 사용자 수를 조회하여 반환.
- **오류 처리** :

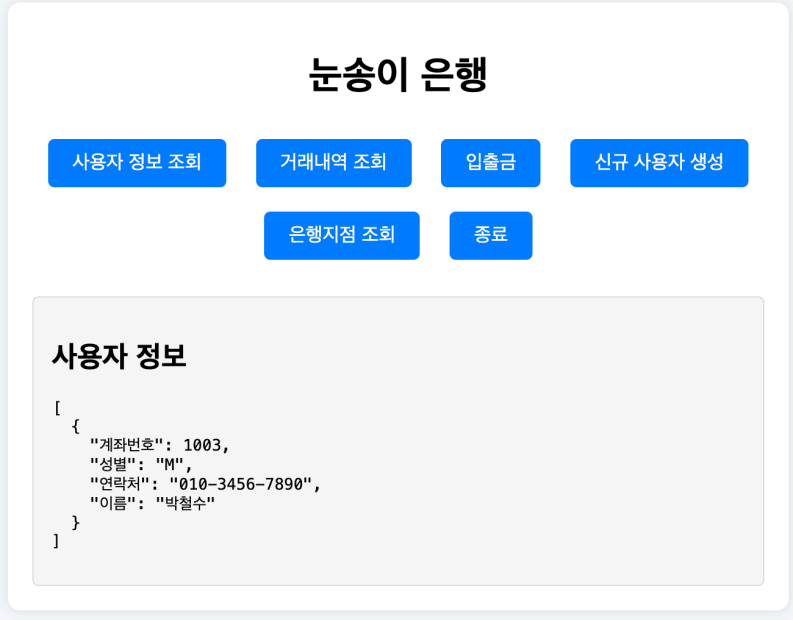
지점 ID 에 해당하는 데이터가 없으면 빈 결과를 반환.
오류 처리는 구현되지 않았지만 SQL 예외 발생 시 서버 로그에 기록.

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

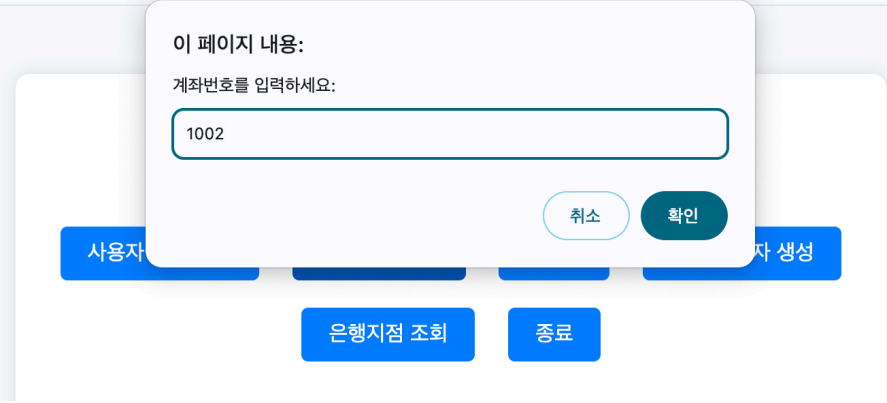
- **디버그 모드**: 디버그 모드로 실행되어 코드 변경 시 자동 재시작 및 상세 오류 메시지 제공.

5. 출력 결과

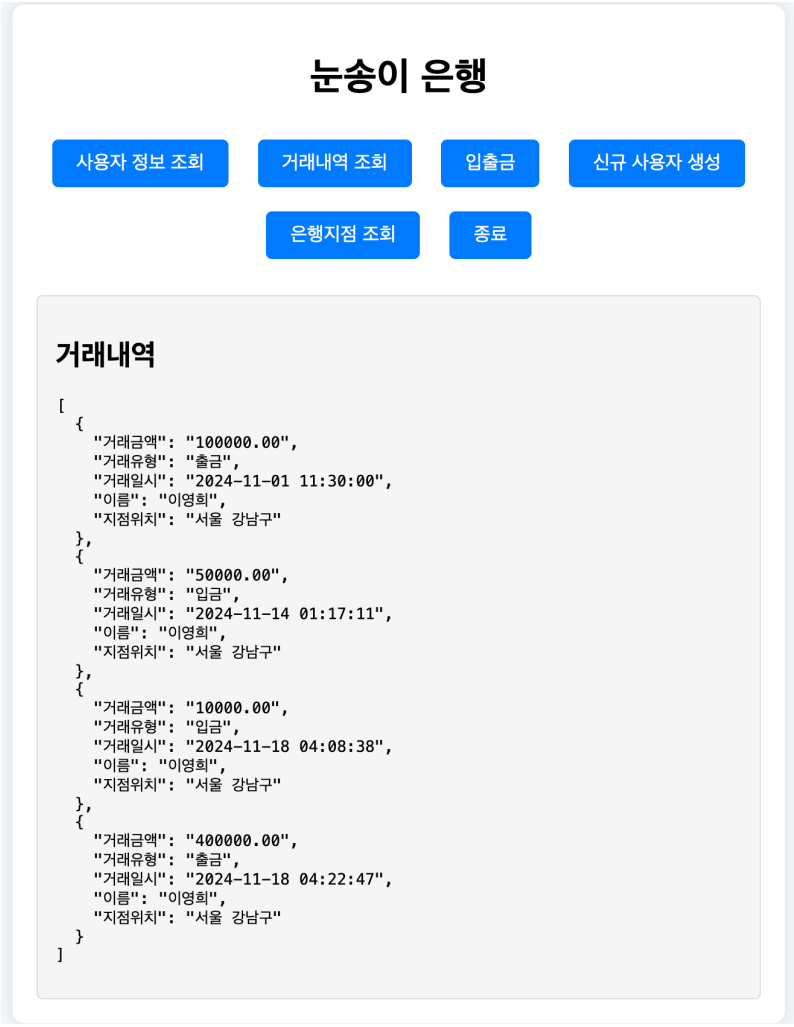
사용자 정보 조회 버튼을 눌렀을 경우입니다.
사용자 아이디를 입력하세요라는 메세지와 함께 입력을 받습니다.



입력한 아이디에 해당하는 사용자의 계좌번호, 성별, 연락처, 이름이 순서대로 출력됩니다.



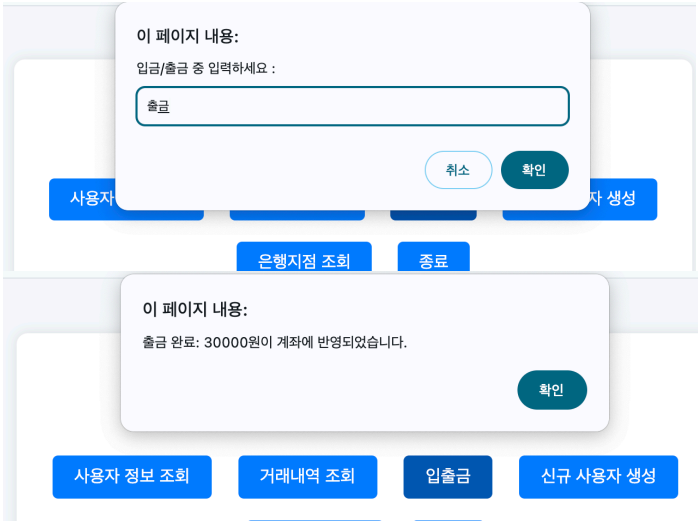
계좌번호 버튼을 입력했을 경우입니다.
계좌번호를 입력받도록합니다.

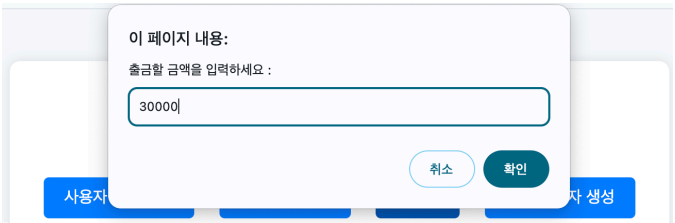


입력한 계좌번호에 해당하는 거래내역이 출력됩니다.

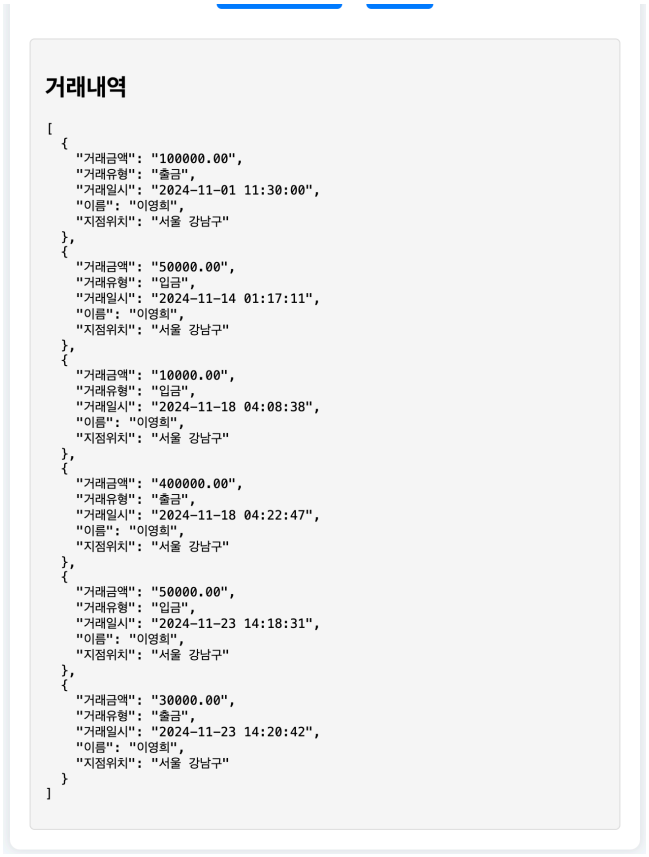
입출금 버튼을 눌렀을 경우입니다.

계좌번호를 입력받고, 입금과 출금중에서 원하는 것을 입력합니다. 해당 예시는 입금하였을 경우입니다. 입금할 금액을 입력받고, 마지막으로 입금완료 메세지가 출력됩니다.





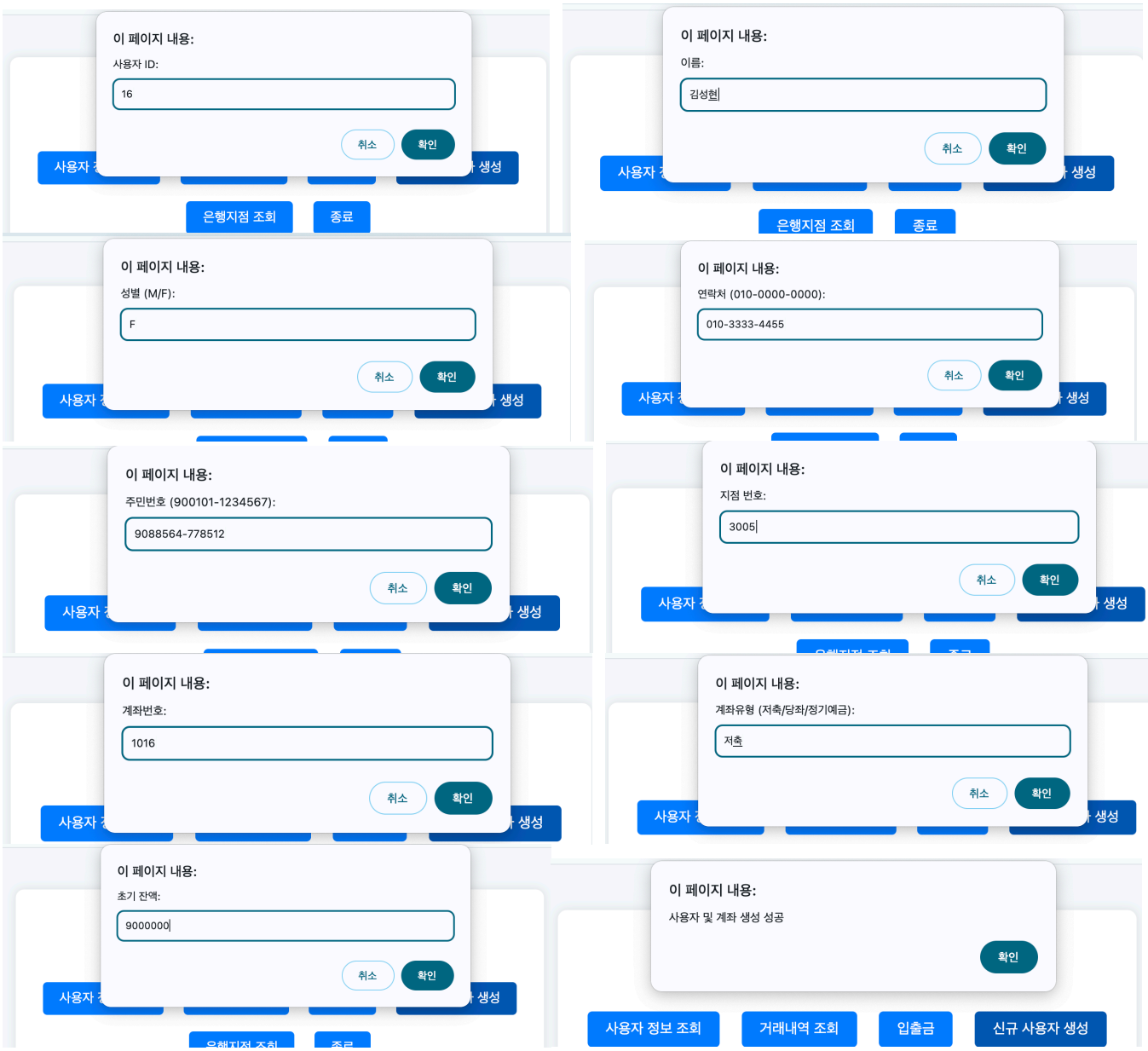
입출금 버튼을 누르고, 출력을 입력했을 경우입니다. 출력할 금액을 입력받고, 마지막으로 입금완료 메세지가 출력됩니다.



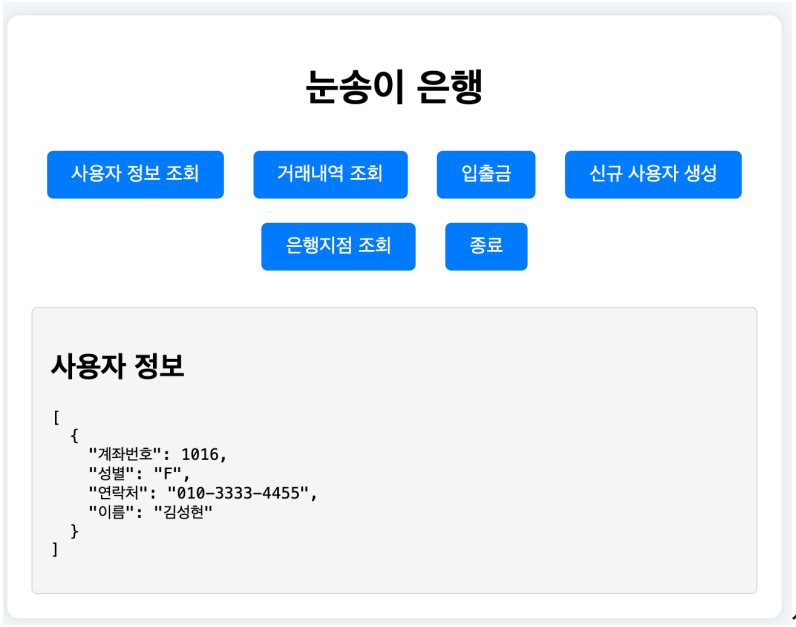
거래내역


```
[
  {
    "거래금액": "100000.00",
    "거래유형": "출금",
    "거래일시": "2024-11-01 11:30:00",
    "이름": "이영희",
    "지점위치": "서울 강남구"
  },
  {
    "거래금액": "50000.00",
    "거래유형": "입금",
    "거래일시": "2024-11-14 01:17:11",
    "이름": "이영희",
    "지점위치": "서울 강남구"
  },
  {
    "거래금액": "10000.00",
    "거래유형": "입금",
    "거래일시": "2024-11-18 04:08:38",
    "이름": "이영희",
    "지점위치": "서울 강남구"
  },
  {
    "거래금액": "400000.00",
    "거래유형": "출금",
    "거래일시": "2024-11-18 04:22:47",
    "이름": "이영희",
    "지점위치": "서울 강남구"
  },
  {
    "거래금액": "50000.00",
    "거래유형": "입금",
    "거래일시": "2024-11-23 14:18:31",
    "이름": "이영희",
    "지점위치": "서울 강남구"
  },
  {
    "거래금액": "30000.00",
    "거래유형": "출금",
    "거래일시": "2024-11-23 14:20:42",
    "이름": "이영희",
    "지점위치": "서울 강남구"
  }
]
```


거래내역 조회기능을 통해 입금과 출금기록이 업데이트된것을 확인할 수 있습니다.

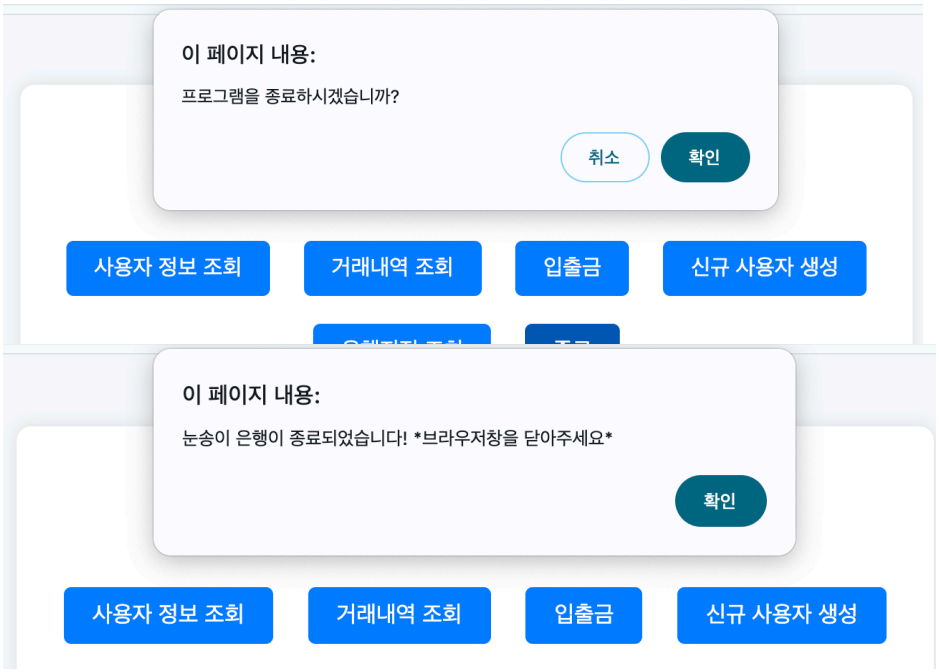


신규 사용자 생성 버튼을 눌렀을 경우입니다.
사용자 아이디, 이름, 성별, 연락처, 주민번호, 지점번호, 계좌번호, 계좌유형, 초기잔액을 입력받습니다.
그리고나서 최종적으로 사용자 및 계좌 생성이 성공되었음을 출력합니다.



사용자 정보조회기능으로 신규 사용자 및 계좌 생성이 되었음을 확인합니다.

은행지점 조회 버튼을 눌렀을 경우입니다.
해당 지점 아이디를 입력받아 지점 정보를 출력합니다. 은행 지점의 사용자 수(고객 수), 위치, 지점장 이름, 직원 수가 출력됩니다.



종료 버튼을 눌렀을 경우입니다.

프로그램 종료를 다시 한번 확인하는 메세지가 출력되고, 확인 버튼을 누르게되면 눈송이 은행이 종료되었다는 메세지가 출력됩니다.

6. Cross check

(시나리오) 고급언어에서 등록한 신규 사용자 및 계좌 확인

```
데이터베이스가 이미 존재합니다.

<<<<<<<<<< 눈송이 은행에 오신걸 환영합니다!! >>>>>
>>>>>

1. 사용자 정보 조회

2. 거래내역 조회

3. 입출금

4. 신규 가입 및 계좌 개설

5. 은행 지점 조회

6. 종료

원하시는 번호를 입력해주세요 : 4
사용자 ID: 17
이름 : 김영진
성별(M/F): M
연락처(010-0000-0000): 010-8888-5544
주민번호(900101-1234567): 9015264-778542
지점 번호 : 3005
계좌번호(4자리): 1017
계좌유형(저축/당좌/정기예금): 정기예금
초기 잔액: 86000000

---------------SQL Code Test---------------
 INSERT INTO 사용자 (사용자_ID, 이름, 성별, 연락처
, 주민번호, 지점_ID) VALUES (%s, %s, %s, %s, %s, %
s);
-------------------------------------------


---------------SQL Code Test---------------
 INSERT INTO 계좌 (계좌번호, 사용자_ID, 계좌유형,
잔액, 지점_ID) VALUES (%s, %s, %s, %s, %s);
-------------------------------------------

새로운 사용자 및 계좌가 성공적으로 등록되었습니다!
```