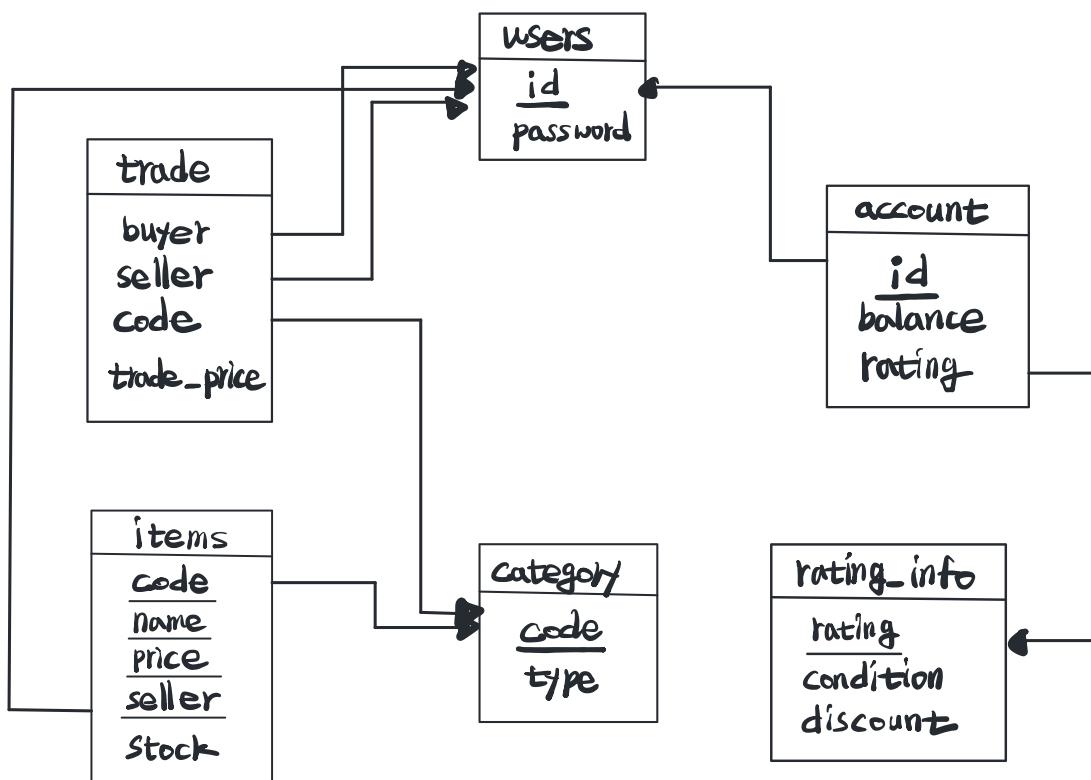


## 1. DB application's Schema Diagram

: 기본적으로 term.sql을 별도의 수정 없이 그대로 사용했다. 다이어그램은 아래와 같다.



## 2. DB application's Functions

```

@app.route('/')
def main():
    return render_template("main.html")

@app.route('/register', methods=['post'])
def register():
    id = request.form["id"]
    password = request.form["password"]
    send = request.form["send"]
    if send == "Login":
        cur.execute("SELECT * FROM users WHERE id = '{}' AND password = '{}'".format(id, password))
        login_result = cur.fetchall()
        if login_result: # admin 추가
            return redirect(url_for('login_action', user_id = id))
        return render_template("login_fail.html")
    elif send == "sign up":
        # need to check redundancy first(ID_collision)
        cur.execute("SELECT * FROM users WHERE id = '{}'".format(id))
        collision_result = cur.fetchall()
        if not collision_result:
            # if there's no collision, proceed to signing up
            cur.execute("INSERT INTO users VALUES('{}', '{}')".format(id, password))
            cur.execute("INSERT INTO account VALUES('{}', 10000, 'beginner')".format(id))
            connect.commit()
        else:
            # there is collision
            return render_template("ID_collision.html")
    return render_template("main.html")

```

- **Main** : DB Trading Site의 초기 화면을 불러오는 함수다. main.html 파일을 렌더링해 로그인 화면을 나타낸다.
- **Register** : 로그인 및 회원가입 기능을 담당하는 함수다. 각각 SQL문을 통해 중복 여부를 판단한다.

- **로그인** : "SELECT \* FROM users WHERE id = '{입력한 id}' AND password = '{입력한 password}';"를 사용하여 DB에 일치하는 id, password가 있으면 'login\_action'함수를 실행하고, 그렇지 않은 경우 login\_fail.html을 렌더링한다.
- **회원가입** : "SELECT \* FROM users WHERE id = '{가입요청 id}';"를 사용해 기존 DB에 가입요청 id가 이미 존재하는지를 파악한다.  
 존재하지 않는 경우, "INSERT INTO users VALUES('{id}', '{password}');"와  
 "INSERT INTO account VALUES('{id}', 10000, 'beginner');;"를 통해  
 users 테이블과 account 테이블에 새로 가입한 회원 정보를 삽입한다.  
 이미 존재하는 경우, ID\_collision.html 파일을 렌더링한다.

**DB Trading Site**

ID: admin  
PASSWORD: 0000

**Admin function**

popular category	best buyer	best seller
electronics	karina	postgres

current user: admin / [Logout](#)  
balance: 10001000  
rating: gold

**ITEMS**

code	name	price	stock	seller	
01	iPad	3000	1	postgres	<input type="button" value="buy"/>
01	Radio	3000	7	wendy	<input type="button" value="buy"/>
00	Database	1000	9	admin	<input type="button" value="buy"/>

**DB Trading Site**

ID: Irene  
PASSWORD: 1111

**term=# select \* from users;**

id	password
admin	0000
postgres	dbdb
joy	1234
karina	1234
winter	aaaa
wendy	2022
Irene	1111

(7 rows)

**term=# select \* from account;**

id	balance	rating
Irene	10000	beginner
joy	1000	beginner
karina	1000	beginner
winter	1000	beginner
postgres	98090	bronze
wendy	13000	beginner
admin	10001000	gold

(7 rows)

```

@app.route('/login_action/<user_id>')
def login_action(user_id):
    cur.execute("SELECT balance, rating FROM account WHERE id = '{}';".format(user_id))
    user_info = cur.fetchall()
    cur.execute("WITH code_cnt(code, cnt) AS (SELECT code, count(code) FROM trade GROUP BY code), \
                max_cnt(value) AS (SELECT max(cnt) FROM code_cnt), \
                max_code(code) AS (SELECT code FROM code_cnt, max_cnt WHERE code_cnt.cnt = max_cnt.value) \
                SELECT type FROM category, max_code WHERE category.code = max_code.code;")
    popular_category = cur.fetchall()
    cur.execute("WITH buying_cnt(buyer,cnt) AS (SELECT buyer,count(buyer) FROM trade GROUP BY buyer), \
                max_cnt(value) AS (SELECT max(cnt) FROM buying_cnt) \
                SELECT buyer FROM buying_cnt, max_cnt WHERE buying_cnt.cnt = max_cnt.value;")
    best_buyer = cur.fetchall()
    cur.execute("WITH selling_cnt(seller,cnt) AS (SELECT seller,count(seller) FROM trade GROUP BY seller), \
                max_cnt(value) AS (SELECT max(cnt) FROM selling_cnt) \
                SELECT seller FROM selling_cnt, max_cnt WHERE selling_cnt.cnt = max_cnt.value;")
    best_seller = cur.fetchall()
    site_info = popular_category + best_buyer + best_seller
    cur.execute("SELECT * FROM items;")
    item_info = cur.fetchall()
    return render_template("login_success.html", id = user_id, u_info = user_info, s_info = site_info, items = item_info)

```

- **Login\_action:** 로그인에 성공했을 때 나타나는 화면을 구현하는 함수다.

WITH, MAX 기능을 사용한 SQL문으로 popular category, best buyer, best seller의 이름을 찾았다.

이 세 가지를 site\_info로 묶어서 html 파일에 전달했다. 마찬가지로 SQL문을 사용해 로그인 유저의 balance 와 rating을 user\_info로, ITEMS에 표시될 정보들을 item\_info로 묶어서 login\_success.html 파일에 넘겨주었다.

id가 admin일 경우에만 특정 기능이 나타나도록 하기 위해서 user\_id도 같이 전달했다.

popular category	best buyer	best seller
electronics	karina	postgres

current user: wendy / [Logout](#)  
balance: 13000  
rating: beginner

**ITEMS**

add					
code	name	price	stock	seller	
01	Ipad	3000	1	postgres	buy
01	Radio	3000	7	wendy	buy
00	Database	1000	9	admin	buy

add					
code	name	price	stock	seller	
01	Ipad	3000	1	postgres	buy
01	Radio	3000	7	wendy	buy
00	Database	1000	9	admin	buy

```

@app.route('/admin_function', methods=['post'])
def admin_function():
    send = request.form["send"]

    cur.execute("SELECT * FROM category;")
    category_list = cur.fetchall()
    cur.execute("SELECT * FROM users;")
    user_list = cur.fetchall()
    cur.execute("SELECT * FROM trade;")
    trade_list = cur.fetchall()
    return render_template('admin_page.html', function = send, categories = category_list, users = user_list, trades = trade_list)

```

- **Admin\_function:** ‘admin’ 의 id로 로그인 했을 때만 사용 가능한 기능을 구현하는 함수입니다.

users info 와 trades info가 기본 기능에 해당하고 나머지 add category와 collect monthly fee는 추가로 구현한 기능입니다. 이 함수에서 사용되는 SQL문은 기본 기능 구현에 필요한 데이터를 전달하는 목적입니다.

나머지 추가 기능에 필요한 SQL 문은 별도의 함수에 따로 구현했습니다. SQL문은 users, trade 각 테이블의 모든 항목을 불러와서 users info, trades info에 표시될 정보들을 가져옵니다(SELECT).

send에 users info 또는 trades info를 받아서 admin\_page.html에 function 값으로 넘겨줍니다.

(users info)

(trades info)

The image shows two separate browser windows. The left window is titled 'admin\_page' and displays a table of user information with columns 'id' and 'password'. The right window is also titled 'admin\_page' and displays a table of trade history with columns 'buyer', 'seller', 'code', and 'trade\_price'. Both tables contain several rows of data.

id	password
admin	0000
postgres	dbdb
joy	1234
karina	1234
winter	aaaa
wendy	2022
Irene	1111

buyer	seller	code	trade_price
postgres	admin	01	1000
joy	postgres	01	3000
karina	postgres	01	3000
karina	postgres	01	3000
karina	postgres	01	3000
winter	postgres	01	3000
winter	postgres	01	6000
wendy	postgres	02	5000
postgres	wendy	01	9000
wendy	admin	00	1000

```
@app.route('/item_add', methods=['post'])
def item_add():
    user_id = request.form["id"]
    cur.execute("SELECT * FROM category;")
    category = cur.fetchall()
    return render_template('item_add.html', category_list = category, id = user_id)
```

- **Item\_add:** ITEMS(현재 거래 가능한 아이템 정보)에 포함될 상품을 추가하는 기능을 구현한 함수입니다.  
카테고리 정보를 표시해주어야 하기 때문에 SQL문을 통해서 category\_list를 불러왔습니다.

```
@app.route('/item_adding', methods=['post'])
def item_adding():
    code = request.form["code"]
    name = request.form["name"]
    price = request.form["price"]
    stock = request.form["stock"]
    seller = request.form["seller"]
    send = request.form["send"]

    cur.execute("SELECT code FROM category WHERE code ='{}';".format(code))
    code_exist = cur.fetchall()
    cur.execute("SELECT id FROM users WHERE id ='{}';".format(seller))
    id_exist = cur.fetchall()

    if send == "add":
        add_success = False
        #check constraints
        if code_exist and id_exist and int(price)>=0 and int(stock)>0:
            cur.execute("SELECT * FROM items WHERE code='{}' AND name='{}' AND price='{}' AND seller = '{}';".format(code, name, price, seller))
            item_exist = cur.fetchall()
            if item_exist:
                cur.execute("UPDATE items SET stock = stock + '{}' WHERE code='{}' AND name='{}' \\\n                AND price='{}' AND seller = '{}';".format(stock, code, name, price, seller))
            else:
                cur.execute("INSERT INTO items VALUES('{}', '{}', '{}', '{}', '{}');".format(code, name, price, stock, seller))
                connect.commit()
                add_success = True
        return render_template('item_add_action.html', success = add_success, id = seller)
    return redirect(url_for('login_action', user_id = seller))
```

The screenshot shows a web application interface. On the left, there is a sidebar titled 'CATEGORY LIST' with a table containing categories 00 through 04. On the right, there is a main form with fields for 'Code', 'Name', 'Price', and 'Stock'. Below the form are 'add' and 'cancel' buttons. The URL in the browser bar is 127.0.0.1:5000/item\_add.

Code
books
electronics
clothing
food
beverage

Code: \_\_\_\_\_  
Name: \_\_\_\_\_  
Price: \_\_\_\_\_  
Stock: \_\_\_\_\_  
add | cancel

- **Item\_adding:** 추가될 상품이 입력된 이후 해당 상품 정보들의 constraint 준수 및 중복 여부를 검사해 items 테이블에 삽입하는 기능을 구현한 함수입니다. SQL문을 코드 순서대로 설명하면 다음과 같습니다. 먼저 코드 번호가 이미 존재하는 것인지 파악하기 위한 SELECT 문입니다. 다음으로는 buyer의 id(id)가 seller의 id(seller)와 일치하는지를 파악하기 위해 사용한 것입니다. 이후 items 테이블의 constraints를 준수하는 tuple에 대하여 기존에 존재하는 tuple은 stock의 수를 추가하는 UPDATE문을 사용했고, 새롭게 추가되는 tuple인 경우 INSERT 문을 사용했습니다.

```
@app.route('/item_buy', methods=['post'])
def item_buy():
    cur_code = request.form["code"]
    cur_name = request.form["name"]
    cur_price = request.form["price"]
    cur_stock = request.form["stock"]
    cur_seller = request.form["seller"]
    user_id = request.form["id"]

    cur.execute("SELECT balance, rating FROM account WHERE id = '{}';".format(user_id))
    user_info = cur.fetchall()

    return render_template('item_buy.html', id = user_id, u_info = user_info, code = cur_code, name = cur_name,
                           price = cur_price, stock = cur_stock, seller=cur_seller)
```

### • Item\_buy:

ITEMS에서 개별 상품 오른쪽에 있는 'buy' 버튼을 눌렀을 때 나타나는 화면을 구현하는 함수입니다. html의 form 기능을 통해서 ITEM의 정보(code, name, price, stock, seller)를 받아와 item\_buy.html 파일로 전달될 수 있도록 했습니다. 또한 유저의 balance, rating 정보가 필요했기 때문에 user\_id를 이용해서 SQL SELECT 문으로 account 테이블에서 해당 정보들을 불러왔습니다.

```
@app.route('/item_buying', methods=['post'])
def item_buying():
    send = request.form["send"]
    quantity = int(request.form["how_many"])
    user_id = request.form["id"]
    user_balance = int(request.form["balance"])
    user_rating = request.form["rating"]
    item_price = int(request.form["price"])
    item_stock = int(request.form["stock"])
    item_code = request.form["code"]
    item_name = request.form["name"]
    seller = request.form["seller"]

    if send == "cancel":
        return redirect(url_for('login_action', user_id = user_id))
    if quantity <= 0:
        return render_template("item_buy_fail.html", id = user_id)
    if user_id == seller:
        return render_template("item_buy_fail.html", cause = "same_id", id = user_id)
    if item_stock < quantity:
        return render_template("item_buy_fail.html", cause = "insufficient_stocks", id = user_id)

    cur.execute("SELECT discount FROM rating_info WHERE rating = '{}'.format(user_rating)")
    discount_percent = cur.fetchall()
    print(float(discount_percent[0][0]))

    total_price = item_price * quantity
    discount_price = total_price * (float(discount_percent[0][0]) / 100)
    final_price = total_price - discount_price
    stock_left = item_stock - quantity

    if user_balance < final_price:
        return render_template("item_buy_fail.html", cause = "insufficient_balance", id = user_id)
    return render_template("item_trade.html", buyer = user_id, seller = seller, balance = user_balance, rating = user_rating, t_price = total_price,
                          dc_price = discount_price, f_price = final_price, stock = item_stock, quantity = quantity, code = item_code, name = item_name)
```

### • ItemBuying:

'/item\_buy'에서 구입할 수량을 입력하고 buy 버튼을 눌렀을 때 나타나는 화면을 구현한 기능입니다.

구입하고자 하는 수량이 0 보다는 커야하고 stock 값 보다는 작아야합니다. 이런 경우에만 rating\_info 테이블에서 user의 rating에 해당하는 discount rate를 SQL로 불러와 total price, discount price, final price를 각각 계산하고 그것을 화면에 표시합니다.

여기서 confirm 버튼을 누르면 item\_trade 함수가 실행됩니다.

code	name	price	stock	seller
01	Ipad	3000	1	postgres

```
@app.route('/item_trade', methods=['post'])
def item_trade():
    send = request.form["send"]
    buyer_id = request.form["buyer"]
    seller_id = request.form["seller"]
    buy_price = request.form["buy_price"].split('.')[0]
    sell_price = request.form["sell_price"].split('.')[0]
    item_code = request.form["item_code"]
    item_name = request.form["item_name"]
    stock = int(request.form["stock"])
    quantity = int(request.form["quantity"])
    item_price = int(sell_price) // quantity
    stock_left = stock - quantity

    if send == "confirm":
        cur.execute("BEGIN;")
        #Balance
        cur.execute("UPDATE account SET balance = balance - '{}' WHERE id = '{}'".format(buy_price, buyer_id))
        cur.execute("UPDATE account SET balance = balance + '{}' WHERE id = '{}'".format(sell_price, seller_id))
        connect.commit()

        #rating
        cur.execute("SELECT balance FROM account WHERE id = '{}'".format(buyer_id))
        buyer_balance = cur.fetchall()
        cur.execute("SELECT rating FROM rating_info WHERE condition <='{}'".format(buyer_balance[0][0]))
        buyer_rating = cur.fetchall()
        cur.execute("UPDATE account SET rating = '{}' WHERE id = '{}'".format(buyer_rating[0][0], buyer_id))
        connect.commit()

        cur.execute("SELECT balance FROM account WHERE id = '{}'".format(seller_id))
        seller_balance = cur.fetchall()
        cur.execute("SELECT rating FROM rating_info WHERE condition <='{}'".format(seller_balance[0][0]))
        seller_rating = cur.fetchall()
        cur.execute("UPDATE account SET rating = '{}' WHERE id = '{}'".format(seller_rating[0][0], seller_id))
        connect.commit()

    #Trade
    cur.execute("INSERT INTO trade VALUES('{}', '{}', '{}', '{}')".format(buyer_id, seller_id, item_code, sell_price))
    connect.commit()

    if stock_left == 0:
        #Delete from items
        cur.execute("DELETE FROM items WHERE code = '{}' AND name = '{}' AND price = '{}' AND seller = '{}'".format(item_code, item_name, item_price, seller_id))
    else:
        #Update stock
        cur.execute("UPDATE items SET stock = '{}' \ WHERE code = '{}' AND name = '{}' AND price = '{}' AND seller = '{}'".format(stock_left, item_code, item_name, item_price, seller_id))
    connect.commit()

return redirect(url_for('login_action', user_id = buyer_id))
```

### • Item\_trade

: confirm 버튼을 눌렀을 때 구매자의 계좌를 차감, 판매자의 계좌를 증가, 각각의 rating 업데이트를 진행합니다. 그리고 거래 내역을 trade 테이블에 기록하고, 만약 남은 stock이 0이 된 경우 items 테이블에서 삭제하는 기능까지 수행합니다.

이때 계좌를 차감, 증가하는 데에는 UPDATE 문이 사용됐고, 거래내역을 기록하는 데는 INSERT 문이 사용됐습니다. 마지막으로 stock이 0이 된 아이템을 삭제하는 데에는 DELETE 문을 이용했습니다.

SELECT문의 경우, rating 갱신을 위해서는 증감된 계좌 값을 commit 이후에 다시 불러와야 하기 때문에 사용됐습니다.

## [ 추가 기능 ]

(1) admin이 카테고리를 추가하는 기능과, (2) admin이 월 수수료(구독료)를 사용자들로부터 수금하는 기능을 추가했습니다.

```
@app.route('/admin_action', methods=['post'])
def admin_action():
    send = request.form["send"]
    code_num = request.form["code"]
    code_type = request.form["type"]

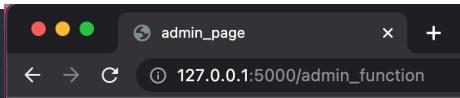
    if send == "cancel":
        return redirect(url_for('login_action', user_id = 'admin'))

    cur.execute("SELECT code FROM category WHERE code = '{}'".format(code_num))
    num_result = cur.fetchall()
    cur.execute("SELECT type FROM category WHERE type = '{}'".format(code_type))
    type_result = cur.fetchall()

    #check constraints
    if not code_num or not code_num.isdigit() or len(code_num)>2:
        return render_template('admin_action_fail', cause='num_invalid')
    if not code_type or code_type.isdigit() or len(code_type)>2:
        return render_template('admin_action_fail', cause='type_invalid')

    #check redundancy
    if num_result:
        return render_template('admin_action_fail', cause='num_collision')
    if type_result:
        return render_template('admin_action_fail', cause='type_collision')

    cur.execute("INSERT INTO category VALUES('{}', '{}')".format(code_num, code_type))
    connect.commit()
    cur.execute("SELECT * FROM category")
    category_list = cur.fetchall()
    return render_template('admin_page.html', function = 'add category', categories = category_list)
```



	CATEGORY LIST
00	books
01	electronics
02	clothing
03	food
04	beverage

Code:

Type:

- **Admin\_action:** category 테이블에서 모든 code와 type을 불러오는 SELECT 문을 사용해서 CATEGORY LIST를 표시했습니다.

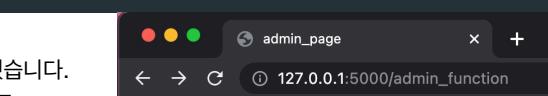
다음으로는 새로운 코드, 타입 값을 입력하는 Input 항목을 만들고 SELECT를 이용해 입력된 값에 대해서 추가 가능한 코드와 타입인지 검사하는 SQL문을 작성했습니다. 추가 가능한 코드, 타입에 대해서 INSERT까지 실행했습니다.

```
@app.route('/admin_collect', methods=['post'])
def admin_collect():
    send = request.form["send"]
    monthly_fee = request.form["fee"]

    if send == 'cancel':
        return redirect(url_for('login_action', user_id = 'admin'))

    cur.execute("SELECT count(id) FROM account;")
    user_cnt = cur.fetchall()
    cur.execute("SELECT id, balance FROM account WHERE balance < {}".format(monthly_fee))
    user_unpaid = cur.fetchall()
    admin_revenue = int(monthly_fee) * (user_cnt[0][0] - len(user_unpaid))

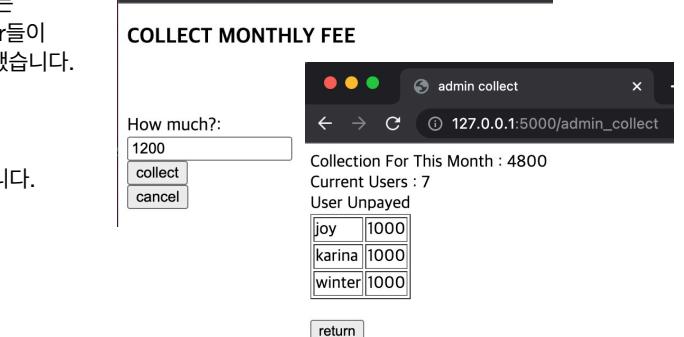
    cur.execute("UPDATE account SET balance = balance - {} WHERE balance >= {}".format(int(monthly_fee), int(monthly_fee)))
    cur.execute("UPDATE account SET balance = balance + {} WHERE id = 'admin'".format(admin_revenue))
    connect.commit()
    return render_template('admin_collect.html', unpaid_user_list = user_unpaid, user_cnt = user_cnt, revenue = admin_revenue )
```



- **Admin\_collect**

: admin이 직접 각 user들에게 걷고자 하는 monthly fee를 입력하도록 했습니다.  
이때 user마다 현재 balance 값을 확인해서 monthly fee를 지불할 수 없는  
unpaid user들이 따로 표시되도록 했습니다. 이들을 제외한 나머지 user들이  
지불한 monthly fee의 총액이 Collection For This Month에 표시되게 했습니다.

user들의 수를 세는 데는 SELECT와 COUNT 가 쓰였고,  
각 user들의 balance에서 mothly fee만큼 차감되고, admin 계좌에  
Collection For This Month가 더해지는 데에는 UPDATE문이 사용됐습니다.



### 3. HTML FILES

- **Main.html**

: 초기 화면에 해당한다. 로그인 또는 회원 가입을 위한 ID와 PASSWORD를 제출하는 submit 타입의 Input들이 존재한다. form을 통해서 register 함수에 전달된다.

- **Login\_success.html**

: if문을 통해서 id가 admin일 때만 특정 항목이 표시되도록 했다. 해당 영역의 input들은 모두 admin\_function 함수로 전달된다. 이외에도 main.html로 돌아가는 로그아웃 버튼을 만들어 두었다. item\_add 함수와 연결되는 add 버튼과 item\_buy 함수와 연결되는 buy 버튼이 있다. hidden을 사용해서 해당 함수들에 필요한 인자(arguments)들을 전달해주었다.

- **Login\_fail.html**

: id 또는 password가 잘못돼 로그인에 실패했을 때 나타나는 화면이다. return 버튼을 통해 main.html로 돌아간다.

- **ID\_collision.html**

: 회원 가입을 요청한 id가 DB상에 이미 존재하는 id일 때 나타나는 페이지이다.

- **Admin\_page.html**

: admin function에서 어떤 버튼을 눌렀느냐에 따라 해당 버튼의 이름이 function 값으로 전달되고, 그에 따라 다른 내용이 표시되는 페이지이다. add category의 경우 category list와 새로운 카테고리 입력부분이 나타난다. collect montly fee일 때는 수금할 요금을 입력하는 페이지가 나타난다. 다음으로 users info의 경우 앞서 admin function 함수에서 받아온 user\_list에 있는 유저 정보들을 for문에 따라 차례로 표에 출력한다. 마지막으로 else 처리된 부분은 trades info로, trade 테이블의 모든 tuple들을 차례로 표에 표시하는 페이지이다.

- **Admin\_action\_fail.html**

: 새로운 카테고리를 추가하는 admin\_action 함수에서 table constraints를 위반하거나 중복되는 코드, 타입이 있는 경우에 표시되는 화면이다. cause 값이 무엇이냐에 따라서 분기되어 코드 또는 타입 가운데 어떤 항목이 잘못 입력된 것인지를 알려준다.

- **Admin\_collect.html**

: 수금이 완료된 이후 결과를 표시하는 페이지이다. 이번달 총 수금액, 현재 사용자 수, 그리고 이번달 요금을 지불하지 않은 유저의 아이디와 계좌정보가 표시된다.

- **Item\_add.html**

: ITEMS에 새로운 아이템을 추가할 때 필요한 코드, 이름, 가격, 수량 정보를 입력하는 페이지이다. 여기서 입력한 정보들이 app.py에 있는 item\_adding 함수로 전달된다.

- **Item\_add\_action.html**

: 새로운 아이템 정보를 입력한 뒤 Add 버튼을 눌렀을 때 성공적으로 입력이 완료가 되면 Successfully Added가, 그렇지 않을 경우 Invalid Request가 출력되는 페이지이다.

- **Item\_buy.html**

: 아이템을 구매하고자 할 때, 몇 개를 살 것인지 구매자의 의사를 입력할 수 있는 페이지이다. 여기서 입력한 quantity 정보가 item\_buying 함수로 전달된다.

- **Item\_buy\_fail.html**

: 아이템 구매 작업이 불가능할 때 나타나는 화면이다. cause 값에 따라 분기가 되어있어서, 어떤 값이 잘못 입력되어 오류가 발생했는지 알 수 있게 해준다.

- **Item\_trade.html**

: 아이템 구매 작업이 가능할 때 최종 confirm을 묻는 페이지이다. confirm 버튼을 누르면 구매정보가 item\_trade 함수로 전달되어 DB 상에 실제 update 작업이 발생한다.