DB assignment 2 Wiki

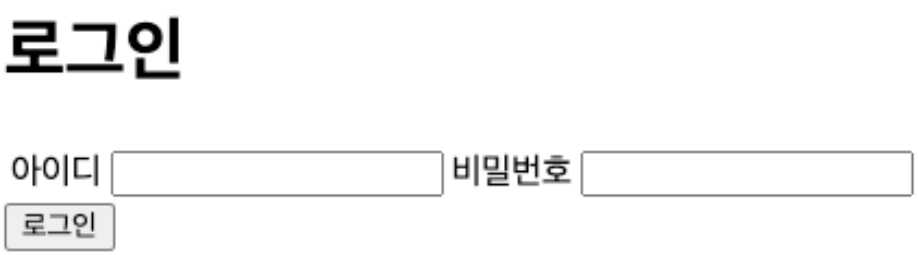
2020018958 송재휘

1. Design

이번 DB assignment2에서 구현해야 하는 simple HTS를 구현하기 위해, 우선 필요한 화면들을 정리했습니다. 명세에서 주어진 기능들이 모두 구현되기 위해서, 정리한 페이지는 다음과 같습니다.

(1) 기본 페이지 ('/') : Simple HTS라는 제목과 로그인, 회원가입을 위한 버튼 추가

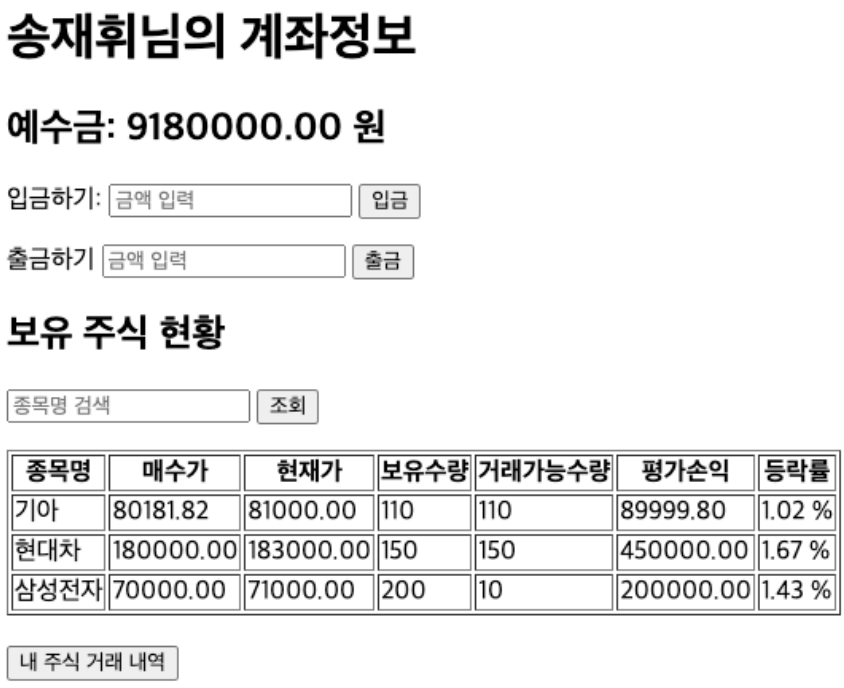
(2) 로그인 페이지 ('/users/login') : 로그인을 위한 페이지 (아이디, 비밀번호 확인)



(3) 회원가입 페이지 ('users/signup') : 회원가입을 위한 페이지 (이름, 아이디, 비밀번호 입력)

(4) 메인 페이지 : 안녕하세요, {이용자 이름}님이라는 문구를 띄우고, 계좌가 없는 경우 계좌 개설 하기 위한 버튼을 보여주고, 버튼을 누른 경우 자동으로 계좌를 생성함. 반대로, 계좌가 있는 경우 내 정보, 주식 정보, 회원 탈퇴, 로그아웃 기능을 위해 버튼을 띄준다. 회원 탈퇴 버튼을 누르면 DB 내의 회원과 관련된 모든 정보가 삭제된다. 로그아웃을 누르면 기본 페이지로 돌아간다.

(5) 내 정보 페이지 ('account/myAccount') : {이용자 이름}님의 계좌 정보라는 문구를 띄우고, 예수금 표시, 밑에는 입금하기, 출금하기를 위한 값 입력 field와 버튼 제공하고, 그 밑에는 보유 주식 현황을 표로 띄워줌. 이때 종목명 검색이 부분 일치 검색이 가능해 종목명을 입력하고 조회를 누르면 밑에 테이블에 보유 주식 정보 검색 결과를 볼 수 있다. 들어가자마자 default로는 현재 보유한 모든 주식 정보를 보여준다. 내 주식 거래 내역 버튼을 누르면 내 주식 거래 내역을 확인하는 페이지로 이동한다.



(6) 내 주식 거래 내역 페이지 ('/account/userTransactionHistory') : 이 페이지에서는 나의 주식 거래 내역을 확인해볼 수 있다. 기간 검색과 종목명 검색 두가지를 지원하는데, 기간 검색을 할 시작 날짜와 끝 날짜를 입력하고 조회 버튼을 누르면 해당 기간 내의 모든 주식 거래 내역을 볼 수 있고, 밑에 종목명 검색의 경우, 기간 검색을 설정하지 않고, 종목명 검색만 할 경우, 해당 종목의 거래 내역 모두를 볼 수 있고, 기간 설정도 하고 종목명 입력도 한 경우엔 해당 기간 내의 해당 종목에 대한 내 주식 거래 내역을 볼 수 있다. 이때, 아래 주식 거래 내역에서 미체결 수량이 0인 경우는 이미 완료된 거래를 의미해 취소 버튼이 없고, 미체결 수량이 있다면 취소 버튼을 통해 주문을 취소할 수 있다.

(7) 주식 정보 페이지 ('/stock/stockList') : 이 페이지에서는 주식 정보를 보여준다. 종목명 검색을 통해 아래 종목 목록에서 종목명에 대한 부분 일치 검색이 가능하다. 별도의 조회를 하지 않은 경우에는, 종목 목록에 현재 DB에 존재하는 모든 주식 종목 정보를 보여준다. 이때, 정보 항목의 보기 버튼을 누르면 해당 주식의 자세한 정보를 볼 수 있는 페이지로 이동한다. 밑에는 주식 통계를 구현하기 위해 만든 거래량 순위이다. 거래량 순위에서는 기간을 입력하고 기간 조회를 누르면 해당 기간 동안 발생한 거래량 정보의 순위를 매겨 보여준다. 이때, 거래량은 체결된 수량을 의미한다. 오늘의 거래량 순위 버튼을 통해 오늘의 거래량 순위를 확인해볼 수도 있다.

주식 정보

종목명 검색: 종목명 입력 조회

종목 목록

종목코드	종목명	정보
000270	기아	보기
000660	SK하이닉스	보기
005380	현대차	보기
005490	POSCO홀딩스	보기
005930	삼성전자	보기
017670	SK텔레콤	보기
035420	NAVER	보기
035720	카카오	보기
055550	신한지주	보기
066570	LG전자	보기
373220	LG에너지솔루션	보기

거래량 순위

기간 검색: 연도. 월. 일. ~ 연도. 월. 일. 기간 조회 오늘의 거래량 순위

순위	종목코드	종목명	거래량
1	035420	NAVER	238424
2	035720	카카오	175175
3	017670	SK텔레콤	158590
4	055550	신한지주	108827
5	066570	LG전자	95779
6	005930	삼성전자	130
7	000270	기아	120
8	005490	POSCO홀딩스	110
9	005380	현대차	100

(8) 주식 상세 정보 페이지 ('/stock/details') : 이 페이지에선 특정 주식의 상세 정보를 보여준다. 우선 페이지 최상단에 종목코드와 종목명을 보여주고, 밑에 종목 정보를 표시한다. 표시하는 종목 정보에는 현재 가격, 전일 종가 대비 주가 등락률이다. 이 두 정보의 경우 명세에 정의된대로 구현할 예정이다. 밑에는 종목 거래 내역을 보여주는데, 이것도 마찬가지로, 기간 검색 기능을 제공한다. 처음엔 아무 정보도 보이지 않고, 기간을 입력한 뒤 조회를 누르면 해당 기간 내에 체결된 거래 내역을 최근 시간 순으로 보여준다. 이때, 밑의 매수, 매도 버튼을 누르면 각각 매수, 매도 페이지로 이동한다.

종목코드: 035720 | 종목명: 카카오

종목 정보

현재 가격: 136000.00

전일 종가 대비 주가 등락률: 1.49%

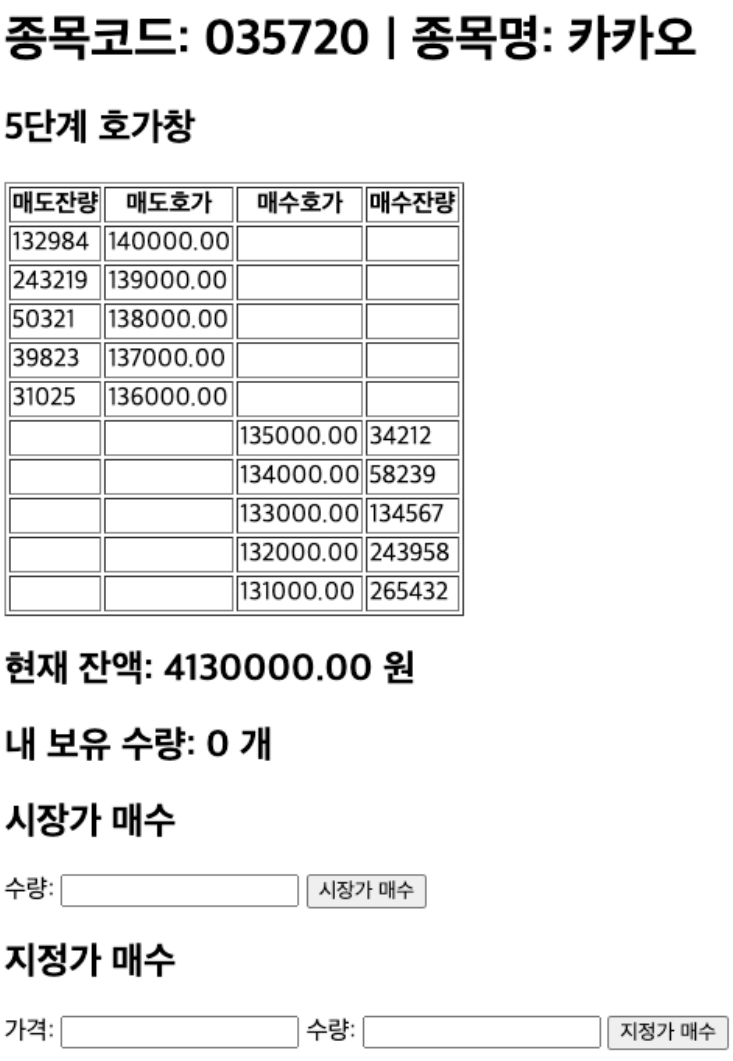
종목 거래 내역

기간 검색: 2024. 11. 01. ~ 2024. 11. 07. 조회

거래 타입	매수/매도가	거래 수량	거래 날짜	거래 시간
BUY	131000.00	379534	2024. 11. 7.	15:40:00
SELL	136000.00	58234	2024. 11. 7.	14:25:00
BUY	132000.00	264539	2024. 11. 7.	13:30:00
SELL	137000.00	63548	2024. 11. 7.	12:50:00
BUY	133000.00	153478	2024. 11. 7.	11:20:00
SELL	138000.00	174562	2024. 11. 7.	10:05:00
BUY	134000.00	72345	2024. 11. 6.	13:20:00
SELL	139000.00	268743	2024. 11. 6.	10:35:00
BUY	135000.00	46234	2024. 11. 6.	09:40:00
SELL	140000.00	257483	2024. 11. 6.	09:10:00

매수 매도

(9) 매수 페이지 ('/transactions/buy') : 이 페이지에서는 실제 해당 종목의 매수 기능을 지원한다. 우선 위에 종목코드와 종목명을 띄우고, 아래에 5단계 호가창을 보여준다. 5단계 호가창은 명세에 나와있는 것처럼 5단계의 매도잔량 및 매도 호가와 매수호가 및 매수잔량을 보여준다. 밑에는 현재 잔액을 보여주고, 해당 주식에 대한 내 보유 수량을 보여준다. 매수 주문 금액이 잔액보다 높은 경우, 매수 주문은 거절된다. 시장가 매수의 경우 수량을 입력하면 가장 낮은 매도호가부터 쭉 긁으면서 매수가 완료되고, 만약 매도 잔량이 없다면 마지막 매수를 한 가격으로 올려놓는다. 이때 5단계 호가창은 매수 주문에 맞게 바뀐다. 지정가 매수의 경우는 가격을 지정해서 매수가 가능한데, 지정가 매수를 존재하는 매도 호가에 대해 수행할 경우 바로 체결되고, 만약 존재하지 않는다면 호가창에 다음 매도 주문을 기다리며 남아있게 된다. 지정가 매수의 경우 만약 가장 낮은 매도 호가보다 높은 가격에 매수를 올리면 지정한 가격으로만 매수가 진행된다. 즉, 이 경우 137000원에 지정가 매수를 올린다면 1000원을 손해보게 된다.



종목코드: 035720 | 종목명: 카카오

5단계 호가창

매도잔량	매도호가	매수호가	매수잔량
132984	140000.00		
243219	139000.00		
50321	138000.00		
39823	137000.00		
31025	136000.00		
		135000.00	34212
		134000.00	58239
		133000.00	134567
		132000.00	243958
		131000.00	265432

현재 잔액: 4130000.00 원

내 보유 수량: 0 개

시장가 매수

수량: 시장가 매수

지정가 매수

가격: 수량: 지정가 매수

(10) 매도 페이지 ('/transactions/sell') : 이 페이지의 경우는 위의 매수 페이지와 동일하다. 차이점은 밑의 시장가, 지정가 매수가 시장가, 지정가 매도로 바뀌었다는 점이다. 매도 방법은 동일하다. 시장가 매도를 한 경우, 가장 높은 매수호가부터 긁으면서 매도가 체결되고, 남는 수량은 마지막 결제한 매도 금액으로 걸려 남아있게 된다. 만약 처음부터 매도 주문이 없었다면, 가장 낮은 매도호가에 걸어둔다. 지정가 매도의 경우는 마찬가지로 가격을 지정하고 매도를 할 수 있는 기능이다. 가장 높은 매수호가보다 낮은 가격으로 지정가 매도를 하는 경우, 똑같이 손해를 보게 되지만 거래는 체결된다. 또 이 페이지에서는 매도를 위해 거래 가능 수량을 보여준다. 만약 거래 가능 수량보다 더 많은 수량을 매도하려고 한다면 해당 주문은 실패한다.

종목코드: 035720 | 종목명: 카카오

5단계 호가창

매도잔량	매도호가	매수호가	매수잔량
132984	140000.00		
243219	139000.00		
50321	138000.00		
39823	137000.00		
31025	136000.00		
		135000.00	34212
		134000.00	58239
		133000.00	134567
		132000.00	243958
		131000.00	265432

현재 잔액: 4130000.00 원

내 보유 수량: 0 개

거래 가능 수량: 0 개

시장가 매도

수량: 시장가 매도

지정가 매도

가격: 수량: 지정가 매도

2. Implementation

구현을 위해 기능을 세분화 하여 각 기능을 처리하기 위한 별도의 .js 파일을 만들고 해당 파일들의 router를 export 해 app.js에서 합쳐서 구현했습니다. 세분화한 카테고리로는 유저 정보 관련 처리를 위한 user.js, 계좌 관련 처리를 위한 account.js, 주식 관련 처리를 위한 stock.js, 주식 거래 관련 처리를 위한 transaction.js, 기본적인 express 설정과 app 실행, view engine 설정, json 정보를 받기 위한 세팅 코드가 포함된 appConfig.js, db와의 연동을 위한 db.js가 있습니다. App.js 파일의 코드는 다음과 같습니다. (appConfig.js와 db.js는 실습에서 해주신 코드와 같습니다.)

```
const setupApp = require('./appConfig');
const app = setupApp();
const userRouter = require('./user');
const accountRouter = require('./account');
const stockRouter = require('./stock');
const transactionRouter = require('./transaction');

app.use('/users', userRouter);
app.use('/account', accountRouter);
app.use('/stock', stockRouter);
app.use('/transactions', transactionRouter);

app.get('/', (req, res) => {
    res.render("app"); // views 폴더에 들어있는 파일명 (.ejs 생략))
});

app.listen(3000, () => {
    console.log("서버 실행 중...");
});
```

지금부터는 페이지 별 구현 과정에 대해 설명하겠습니다. 실제 모습은 위의 Design 파트에 첨부했습니다. 모든 코드를 설명하기에는 코드의 양이 너무 많아 핵심적인 코드 위주로 설명하고자 합니다.

(1) 기본 페이지 ('/')

Localhost:3000이라는 url에 접속하게 되면, app.js에 있는 app.get('/')이라는 코드 부분이 실행되게됩니다. 해당 코드에서는 그저 app.ejs파일을 렌더링합니다. 기본 페이지의 경우 app.ejs에 해당 페이지의 내용이 정리되어 있습니다. 우선, 로그인 버튼을 누르게 되면 로그인을 처리하는 '/users/login'에 해당하는 백엔드 코드로 이동하게 됩니다. 반대로, 회원가입을 누를 경우 회원가입을 처리하는 'users/signup'에 해당하는 백엔드 코드로 이동하게 됩니다. 코드는 다음과 같습니다.

```
<script>
      const loginButton = document.getElementById("login");
      const signupButton = document.getElementById("signup");

      loginButton.addEventListener('click', () => {
         window.location = 'http://localhost:3000/users/login';
      });
      signupButton.addEventListener('click', () => {
         window.location = 'http://localhost:3000/users/signup';
      });
```

```
    </script>
```

(2) 로그인 페이지 ('/users/login')

기본 페이지에서 로그인 버튼을 누르면 로그인 페이지의 url과 관련된 user.js의 코드가 실행됩니다. 만약, 단순히 해당 페이지에 접근하기 위한 get의 경우에는 loginPage.ejs파일을 렌더링 하고, 아이디와 비밀번호가 입력하고, 로그인 버튼을 눌렀다면, post 요청이 날아오게 되고, 이때 SQL문을 통해 입력된 userID와 userPassword 정보를 DB의 User 테이블의 username, password와 일치하는 tuple이 있는지 쿼리를 날립니다.

```
const [userResults] = await sql_connection.promise().query(
          `SELECT *
           FROM User
           WHERE username = ? and password = ?`,
           [userID, userPassword]
       );
```

이후 만약 해당 정보가 있다면 (userResults.length > 0), 해당 유저가 계좌 정보가 있는지 확인하기 위해 DB의 Account 테이블에 해당 user_id와 일치하는 계좌 정보가 있는지 확인하는 query를 날립니다. 그렇지 않다면, 'fail'이라는 값이 담긴 loginResult를 loginPage.ejs에 제공하면서 렌더링합니다. loginPage.ejs파일에서는 만약 해당 loginResult에 'fail'이 들어있을 경우, 로그인 실패라는 alert를 띄웁니다.

```
const [accountResults] = await sql_connection.promise().query(
              `SELECT *
               FROM Account
               WHERE user_id = ?`,
              [userResults[0].user_id]
          );
```

이후 만약 해당 정보가 있다면 (accountResults.length > 0), mainPage.ejs파일을 렌더링 하는데 이때 인자로 clientName, userID, accountID 정보를 제공합니다. 반대로 해당 정보가 없다면 clientName과 userID 정보만을 제공합니다.

(3) 회원가입 페이지 ('users/signup')

기본 페이지에서 회원가입 버튼을 누른 경우 user.js의 '/signup'이라는 url로의 요청을 처리하는 부분의 코드가 실행됩니다. 로그인과 마찬가지로, 처음 접근하는 경우, 단순히 signupPage.ejs파일을 렌더링합니다. 위의 사진에서 확인할 수 있는 것처럼 signupPage.ejs의 경우, 이름, 아이디, 비밀번호를 입력할 수 있는 input 필드가 존재하고 값을 입력 후 회원가입을 버튼을 누르게 되면, 해당 form 정보를 포함한 채 같은 페이지에 post 요청을 하게 됩니다. 이때 백엔드 코드에서는 req.body에 존재하는 form의 정보인 이름, 아이디, 비밀번호 정보를 읽고, 해당 아이디가 DB에 존재하는지 확인합니다.

```
sql_connection.query('SELECT * FROM User WHERE username = ?', [userID] …
```

만약 존재한다면 (results.length > 0), signupResult라는 변수에 'fail'값을 전달하고 signupPage.ejs를 다시 렌더링 합니다. 이때 signupPage.ejs에서 signupResult라는 변수에 'fail'값이 존재하는 경우, 이미 존재하는 아이디라는 alert를 띄웁니다. 만약 중복된 아이디가 DB에 없다면, DB의 user 테이블에 해당 이름, 아이디, 비밀번호에 대한 새로운 튜플을 추가합니다. 이후 로그인 페이지로 이동합니다.

```
sql_connection.query('INSERT INTO User(name, username, password) values (?, ?, ?)',
[name, userID, userPassword], (error, results, fields) => {
            if (error) throw error;
            else {
                res.redirect('/users/login');
            }
```

(4) 메인 페이지

로그인 페이지에서 로그인을 성공하게 되면 메인 페이지로 이동합니다. 로그인 페이지에서 넘긴 변수 중 accountID 값이 있는지 확인하고, 해당 값이 있는 경우에는 내 정보, 주식 정보, 회원 탈퇴, 로그아웃 버튼을 띄우고, 반대로 해당 값이 undefined인 경우에는 계좌 계설하기라는 버튼을 보여줍니다. 이때, 각각의 버튼은 해당 기능을 담당하는 url로 이동시키고, 해당 url과 관련된 백엔드 코드가 실행되게 됩니다. 내 정보의 경우와 주식 정보 버튼의 경우 처리 과정에서 필요한 정보로 각각 account_id와 user_id, account_id와 clientName를 넘기고, 회원 탈퇴의 경우 userID 값을, 계좌 개설하기의 경우 userID와 clientName 값을 함께 넘깁니다.

```
<% if (typeof accountID !== 'undefined') { %>
      <button id="myAccount">내 정보</button>
      <button id="stockInfo">주식 정보</button>
      <button id="deleteUser">회원 탈퇴</button>
      <button id="logout">로그아웃</button>

      <script>
         const accountButton = document.getElementById("myAccount");
         const stockInfoButton = document.getElementById("stockInfo");
         const deleteButton = document.getElementById("deleteUser");
         const logOutButton = document.getElementById("logout");

         accountButton.addEventListener('click', () => {
            window.location =
'http://localhost:3000/account/myAccount?account_id=<%= accountID %>&user_id=<%=
userID %>';
         });
         stockInfoButton.addEventListener('click', () => {
            window.location =
'http://localhost:3000/stock/stockList?account_id=<%= accountID %>&clientName=<%=
clientName %>';
         });
         deleteButton.addEventListener('click', () => {
```

```
            window.location = 'http://localhost:3000/users/deleteUser?user_id=<%=
userID %>';
         })
         logOutButton.addEventListener('click', () => {
            window.location = 'http://localhost:3000/';
         })
      </script>
   <% } else { %>
      <button id="createAccount">계좌 개설하기</button>
      <script>
         const createButton = document.getElementById("createAccount");

         createButton.addEventListener('click', () => {
            window.location =
"http://localhost:3000/account/createAccount?user_id=<%= userID %>&clientName=<%=
clientName %>";
         });
      </script>
   <% } %>
```

(5) 내 정보 페이지 ('account/myAccount')

내 정보 페이지의 url로 접근이 발생하면, 해당 접근을 처리하기 위해 백엔드 코드가 실행됩니다. 백엔드 코드에선 먼저 메인 페이지에서 넘긴 user_id와 account_id 값을 읽습니다. 이후, 사용자 정보 및 계좌 정보 조회를 위해 sql query를 날립니다. 우선. user_id에 해당하는 user테이블의 튜플을 읽기 위해 아래 코드를 실행합니다.

```
const [userResult] = await sql_connection.promise().query('SELECT * FROM User WHERE
user_id = ?', [userID]);
```

두번째로, 해당 account_id에 해당하는 account 테이블의 튜플을 읽기 위해 query를 날립니다.

```
const [accountResult] = await sql_connection.promise().query('SELECT * FROM Account
WHERE account_id = ?', [accountId]);
```

세번째로, 해당 주식 계좌의 보유주식을 읽기 위해 Owning_stock과 Stock을 join해서 해당 주식 계좌에서 보유 중인 주식의 종목코드와 이름, 수량, 평단가를 읽어옵니다.

```
const [owningStocks] = await sql_connection.promise().query(
         `SELECT O.stock_id, S.name, O.quantity, O.buying_price
          FROM Owning_stock as O
          JOIN Stock as S ON O.stock_id = S.stock_id
          WHERE O.account_id = ?`,
         [accountId]
      );
```

이후, 보유한 주식 정보가 담긴 owningStocks에서 주식을 한 종목 씩 읽으면서 현재가, 거래가능수량, 평가손익, 등락률을 차례로 계산합니다. 이후 만약 입력 받은 user_id와 account_id에 해당하는 유저 튜플과 계좌 튜플이 존재한다면, myAccount.ejs를 렌더링하고 인자로 clientName과 해당 계좌 정보, 보유 주식 정보를 제공합니다. 이때 현재가는 매도 호가 중 가장 낮은 값과 매수

호가 중 가장 높은 값 중 가장 최근에 거래된 값에 의해 결정되는데 이를 위해 다음과 같은 쿼리를 사용했습니다. 다음 쿼리는 가장 낮은 매도호가와 가장 높은 매수 호가를 읽고, 해당 거래의 시간을 비교해 더 최근의 것을 현재가로 사용합니다. 마지막 코드는 거래가 없어서 recent_price가 결정되지 않는 경우를 대비해 평단가를 현재 가격으로 설정합니다.

```
// 현재가 계산 (매도 호가 중 가장 낮은 값과 매수 호가 중 가장 높은 값 중 최근 거래가로 설정)
          const [[{ recent_price }]] = await sql_connection.promise().query(
              `SELECT
                  CASE
                      WHEN sell_trade.date > buy_trade.date OR (sell_trade.date =
buy_trade.date AND sell_trade.time > buy_trade.time)
                      THEN sell_trade.price
                      ELSE buy_trade.price
                  END as recent_price
              FROM
                  (SELECT price, date, time
                  FROM Transaction
                  WHERE stock_id = ? and type = 'SELL' and left_quantity > 0
                  ORDER BY price
                  LIMIT 1) as sell_trade,
                  (SELECT price, date, time
                  FROM Transaction
                  WHERE stock_id = ? and type = 'BUY' and left_quantity > 0
                  ORDER BY price desc
                  LIMIT 1) as buy_trade`,
              [stockId, stockId]
          );

        stock.current_price = recent_price || stock.buying_price;
```

이후 거래 가능 수량 계산을 하기 위해서는 현재 매도 주문이 중에서 미체결 수량을 봐야하는데, 이를 확인하기 위해, 다음과 같은 쿼리를 사용했습니다. 해당 계좌의 해당 종목코드의 주식 주문 중 매도 타입이고, 미체결 수량이 있는 경우 해당 미체결 수량을 모두 더 해 pending_quantity라는 값을 읽어옵니다.

```
// 거래 가능 수량 계산
          const [pendingSellOrders] = await sql_connection.promise().query(
              `SELECT coalesce(sum(left_quantity), 0) as pending_quantity
              FROM Transaction
              WHERE account_id = ? and stock_id = ? and type = 'SELL' and
left_quantity > 0`,
              [accountId, stockId]
          );
```

이후, 현재 보유 주식에서 해당 값을 빼, 거래 가능 수량을 계산합니다.

평가 손익 계산과 등락률 계산의 경우 명세의 공식을 이용했습니다.

```
// 평가 손익 계산
          stock.profit_loss = ((stock.current_price - stock.buying_price) *
stock.quantity).toFixed(2);

          // 등락률 계산
          stock.price_change_rate = (((stock.current_price - stock.buying_price) /
stock.buying_price) * 100).toFixed(2);
```

이후 myAccount.ejs 파일에서는 렌더링 하면서 보낸 clientName, account, owning_stock 정보를 사용해 이용자 이름, 예수금, 종목명, 매수가, 현재가, 보유수량, 거래가능수량, 평가손익, 등락률을 보여줍니다.

또한 해당 페이지에서는 예수금의 입금 및 출금 기능도 제공하는데, 이를 위해 입금 input 필드에 값을 입력하고, 입금하기 버튼을 누르면 백엔드 코드의 ('/deposit') 부분이 실행되게 되고 해당 부분에서 계좌 예수금의 증가를 처리하기 위해 다음과 같은 sql query를 사용합니다.

```
await sql_connection.promise().query(
          `UPDATE Account
           SET balance = balance + ?
           WHERE account_id = ?`,
          [amount, account_id]);
```

반대로, 매도하기 버튼을 누른 경우, 백엔드 코드의 ('/withdraw') 부분이 실행되고, 예수금의 감소를 처리하기 위해 다음과 같은 sql 쿼리를 사용합니다.

```
await sql_connection.promise().query(
          `UPDATE Account
           SET balance = balance - ?
           WHERE account_id = ?`,
          [amount, account_id]);
```

마지막으로, myAccount 페이지에서는 내 보유 주식에 대한 종목명 부분 일치 검색 기능을 제공하는데, 종목명을 입력하고, 조회 버튼을 누르면 백엔드 코드의 searchStock관련 코드가 실행되고, 해당 코드에서 부분일치 검색을 위해 아래와 같은 sql 쿼리를 사용한다. 이후 현재가, 거래가능수량, 평가손익, 등락률을 구하는 코드는 위와 같다.

```
const [owningStocks] = await sql_connection.promise().query(
          `SELECT O.stock_id, S.name, O.quantity, O.buying_price
           FROM Owning_stock as O
           JOIN Stock as S ON O.stock_id = S.stock_id
           WHERE O.account_id = ? AND S.name LIKE ?`,
          [account_id, `%${stockName}%`]
      );
```

마지막으로 밑에 존재하는 내 주식 거래 내역 버튼을 누르면 account_id 정보와 함께 '/account/userTransactionHistory' url로 이동한다.

```
<button
onclick="window.location.href='/account/userTransactionHistory?account_id=<%=
account.account_id %>'">내 주식 거래 내역</button>
```

(6) 내 주식 거래 내역 페이지 ('/account/userTransactionHistory')

이렇게 내 주식 거래 내역 url로 접근하게 된 경우, 관련 url을 처리하는 백엔드 코드가 실행된다. 위 코드에서는 아까 넘겨준 account_id를 읽고, userTransactionHistory.ejs파일을 해당 값을 넘겨주며 렌더링한다. 해당 페이지에는 내 주식 거래 내역의 종목명, 거래, 타입, 매수/매도가, 거래 수량, 미체결 수량, 거래 날짜, 거래 시간, 취소 등을 확인할 수 있는데 이때 기간 검색과 종목명 일치 검색이 둘다 가능하다. 기간을 설정하고 조회 버튼을 누르면 '해당경로/filter'라는 url을 처리하는 백엔드 코드로 이동하고 해당 날짜 사이의 정보를 읽어오기 위해 다음과 같은 sql 쿼리문을 사용해 처리 후 해당 값을 제공한다. 이 코드는 해당 계좌의 거래날짜인 date가 입력된 두 기간 사이인 튜플들을 읽어오는 쿼리이다.

```
const [transactions] = await sql_connection.promise().query(
          `SELECT S.name, T.type, T.price, T.quantity, T.date, T.time,
T.left_quantity, T.transaction_id
           FROM Transaction as T
           JOIN Stock as S ON T.stock_id = S.stock_id
           WHERE T.account_id = ? AND T.date BETWEEN ? AND ?
           ORDER BY T.date desc, T.time desc`,
          [account_id, startDate, endDate]
       );
```

종목명 검색의 경우에는 만약 기간 설정이 되어있지 않다면, 종목명 부분 일치 검색만을 수행하고, 해당 기능을 위한 sql 쿼리는 다음과 같다. 뒤에서 params를 활용해 ?에 해당하는 값을 전달한다.

```
query = `
              SELECT S.name, T.type, T.price, T.quantity, T.date, T.time,
T.left_quantity, T.transaction_id
              FROM Transaction as T
              JOIN Stock as S ON T.stock_id = S.stock_id
              WHERE T.account_id = ? AND S.name LIKE ?
              ORDER BY T.date desc, T.time desc
          `;
```

반대로, 기간이 있는 경우 해당 기간 내의 종목명 부분일치 검색을 실행한다. 위 쿼리는 다음과 같다.

```
// 기간이 설정된 경우에만 기간 조건 추가
       if (startDate && endDate) {
          query = `
              SELECT S.name, T.type, T.price, T.quantity, T.date, T.time,
T.left_quantity, T.transaction_id
              FROM Transaction as T
              JOIN Stock as S ON T.stock_id = S.stock_id
              WHERE T.account_id = ? and S.name LIKE ? and T.date BETWEEN ? and ?
              ORDER BY T.date desc, T.time desc
          `;
          params.push(startDate, endDate);
```

(7) 주식 정보 페이지 ('/stock/stockList')

이번엔 메인 페이지에서 주식 정보라는 버튼을 누른 경우이다. 버튼을 누르게 되면 해당 url을 처리하는 백엔드 코드로 이동하게 된다. 해당 코드에서는 단순히 clientName과 account_id 값을 넘기며 stockList.ejs 파일을 렌더링 한다. 이 페이지에서는 우선 DB에 존재하는 모든 종목 목록을 보여준다. 이를 처리하기 위한 코드는 다음과 같다.

```
// 페이지 로드 시 전체 주식 목록 표시
      document.addEventListener("DOMContentLoaded", searchStocks);
```

searchStocks라는 함수는 종목명 부분 일치 검색에서도 사용하는 함수인데, 이 함수에서는 '/stock/search'라는 url에 해당하는 백엔드 코드를 호출하고 해당 코드에서는 만약 부분 일치 검색을 위한 종목명이 입력 되었다면, 해당 이름을 사용해 stock 테이블에서 튜플을 읽고 아니라면 stock 테이블의 모든 튜플을 읽어서 띄워준다. 코드는 다음과 같다.

```
const query = stockName
          ? `SELECT * FROM Stock WHERE name LIKE ?`
          : `SELECT * FROM Stock`;
      const params = stockName ? [`%${stockName}%`] : [];

      const [stocks] = await sql_connection.promise().query(query, params);
      res.json(stocks);
```

그리고 종목 옆에는 보기라는 버튼이 있는데 해당 버튼을 누르면 stock의 상세 페이지인 '/stock/details'로 이동한다. 밑에는 주식 통계를 보여주기 위해 선택한 거래량 순위 정보를 띄워주도록 했는데, 거래량 순위도 기간 검색이 가능하다. 기간 검색 기능과 오늘의 거래량 순위 확인 기능이 둘다 있는데, 만약 오늘의 거래량 순위 버튼을 누르면 'today'라는 인자가 함수로 전달된다. 반대로 기간을 설정하고 조회를 누르면, 인자 없이 showTransactionRanking이라는 함수가 실행 되고 해당 함수에서는 인자가 'today'라면 시작과 끝 기간을 오늘로 설정한다. 이후 '/stock/transaction-ranking'이라는 url로 이동하고, 해당 url을 처리하는 백엔드 코드에서는 특정 기간 동안의 거래량 순위를 계산하기 위해 매수 정보를 읽은 거래량 랭킹을 내림차순으로 정렬해 뽑고, 매도 정보를 읽은 거래량 랭킹을 내림차순으로 정렬해 읽는다. 이때 거래량은 T.left_quantity가이거나 T.quantity > T.left_quantity인 튜플을 대상으로 한다. 코드는 다음과 같다.

```
// 특정 기간 동안의 거래량 순위를 조회
      const [transactionBuyRanking] = await sql_connection.promise().query(
          `SELECT S.stock_id, S.name, sum(T.quantity - T.left_quantity) as
total_volume
           FROM Transaction as T
           JOIN Stock as S ON T.stock_id = S.stock_id
           WHERE T.date BETWEEN ? and ? and T.type = 'BUY'
             and (T.left_quantity = 0 or T.quantity > T.left_quantity)
           GROUP BY S.stock_id, S.name
           ORDER BY total_volume desc`,
          [startDate, endDate]
      );
```

```
        const [transactionSellRanking] = await sql_connection.promise().query(
            `SELECT S.stock_id, S.name, sum(T.quantity - T.left_quantity) as
total_volume
             FROM Transaction as T
             JOIN Stock as S ON T.stock_id = S.stock_id
             WHERE T.date BETWEEN ? and ? and T.type = 'SELL'
                and (T.left_quantity = 0 or T.quantity > T.left_quantity)
             GROUP BY S.stock_id, S.name
             ORDER BY total_volume desc`,
            [startDate, endDate]
        );
```

이후, 이렇게 두개를 뽑은 이유는 거래량을 계산 하는 과정에서, 매수 주문의 경우 옛날에 걸어놓은 주문에 대해 매도가 이루어진다면 Date 범위에 속하지 않기 때문에 정확한 거래량 측정이 어렵다는 이유이다. 따라서 두 값을 뽑고, 특정 주식에 대해 거래량이 더 많은 값을 선택한다. 만약 한 주식이 한 곳에서만 나온다면, 이를 바로 사용한다. 이 처리를 위한 코드는 다음과 같다. 이렇게 값을 수정해 거래량 정보를 뽑고 이를 내림차순으로 정렬한다.

```
// 거래량 계산 (매수, 매도 내역 기준 높은 값으로 설정)
        const buyMap = Object.fromEntries(transactionBuyRanking.map(item =>
[item.stock_id, item]));
        const sellMap = Object.fromEntries(transactionSellRanking.map(item =>
[item.stock_id, item]));

        const transactionRanking = Object.values(
            (transactionBuyRanking.length > transactionSellRanking.length) ? buyMap :
sellMap
        ).map(item => {
            const stockID = item.stock_id;
            const buyVolume = buyMap[stockID] ?
Number(buyMap[stockID].total_volume) : 0;
            const sellVolume = sellMap[stockID] ?
Number(sellMap[stockID].total_volume) : 0;

            // 더 큰 값 사용
            const maxVolume = Math.max(buyVolume, sellVolume);

            return {
                stock_id: stockID,
                name: item.name,
                total_volume: maxVolume
            };
        });

        transactionRanking.sort((a, b) => {
            // 거래량 같은 경우 이름 오름차순으로 정렬
            if (b.total_volume === a.total_volume) {
                return a.name.localeCompare(b.name);
            }
            // 거래량 다른 경우, 거래량 내림차순으로 정렬
```

```
            return b.total_volume - a.total_volume;
        });
```

이후 이정보를 stockList.ejs 파일에 보내고, 해당 파일에서는 이 정보를 읽어 랭킹을 표시하게 된다.

(8) 주식 상세 정보 페이지 ('/stock/details')

주식 정보 페이지의 종목 목록에서 보기 버튼을 누르면 종목 코드, 이용자 이름, account_id와 함께 '/stock/details'라는 url로 이동하는데 해당 url을 처리하는 백엔드 코드가 실행된다. 해당 코드에서는 위 값들을 읽고, 종목명 및 현재가 계산을 진행한다. 우선 종목명을 읽기 위해 다음과 같은 sql query를 실행한다.

```
const [[stockInfo]] = await sql_connection.promise().query(
            `SELECT stock_id, name
             FROM Stock
             WHERE stock_id = ?`,
            [stock_id]
        );
```

현재가의 경우 앞에서 설명한 코드와 동일하다.

```
// 현재가 계산 (매도 호가 중 가장 낮은 값과 매수 호가 중 가장 높은 값 중 최근 거래가로 설정)
        const [[{ recent_price }]] = await sql_connection.promise().query(
            `SELECT
               CASE
                  WHEN sell_trade.date > buy_trade.date OR (sell_trade.date =
buy_trade.date AND sell_trade.time > buy_trade.time)
                  THEN sell_trade.price
                  ELSE buy_trade.price
               END as recent_price
             FROM
               (SELECT price, date, time
                FROM Transaction
                WHERE stock_id = ? and type = 'SELL' and left_quantity > 0
                ORDER BY price
                LIMIT 1) as sell_trade,
               (SELECT price, date, time
                FROM Transaction
                WHERE stock_id = ? and type = 'BUY' and left_quantity > 0
                ORDER BY price desc
                LIMIT 1) as buy_trade`,
            [stock_id, stock_id]
        );

        if (recent_price === undefined) {
            recent_price = 0;
        }
```

또한 여기선 전일 종가 대비 주가 등락률을 표시하기 위해 전일 종가를 계산하는데 주문의 date가 어제인 주문 중에서 가장 마지막에 체결된 가격을 전일 종가로 설정한다. 이를 뽑는 sql query는 다음과 같다.

```
// 전일 종가 계산 (어제 마지막 거래 가격)
        const [[{ previous_close_price }]] = await sql_connection.promise().query(
            `SELECT T.price as previous_close_price
             FROM Transaction as T
             WHERE T.stock_id = ? AND T.date = CURDATE() - INTERVAL 1 DAY
             ORDER BY T.time DESC
             LIMIT 1`,
            [stock_id]
        );
```

이 정보를 활용해 현재가에서 빼고, 전일 종가로 나눈 뒤 100을 곱해 전일 종가 대비 주가 등락률을 계산한다

```
// 등락률 계산
        let priceChangeRate = null;
        if (previous_close_price) {
            priceChangeRate = ((recent_price - previous_close_price) /
previous_close_price) * 100;
        }
```

이후, stockDetails.ejs 페이지를 렌더링한다. 해당 페이지에서는 종목 거래내역을 보여주고, 기간 필터로 거래 내역을 검색할 수 있는데, 기간 설정 후 조회를 누르면 '/stock/details/transactions'로 이동한다. 해당 url을 처리하는 백엔드 코드에서는 해당 기간 동안의 거래 내역을 최근 순서대로 읽기 위해 다음과 같은 쿼리를 실행한다. 기간 검색의 쿼리는 앞과 비슷하다.

```
`SELECT type, price, quantity, date, time
             FROM Transaction
             WHERE stock_id = ? AND date BETWEEN ? AND ?
             ORDER BY date DESC, time DESC`,
            [stock_id, startDate, endDate]
        );
```

이후 이 정보를 .ejs 파일에 전달하고, 해당 파일에서는 위의 거래 내역을 보여준다.

마지막으로 해당 페이지의 맨 아래 쪽에는 매수, 매도라는 버튼이 있는데 해당 버튼을 누르면, 매수 그리고 매도 페이지로 이동한다. 해당 코드는 다음과 같다.

```
// 매수 페이지로 이동
        function goToBuyPage() {
            window.location.href =
`/transactions/buy?stock_id=${stockID}&clientName=${clientName}&account_id=${accoun
t_id}`;
        }

        function goToSellPage() {
            window.location.href =
`/transactions/sell?stock_id=${stockID}&clientName=${clientName}&account_id=${accou
nt_id}`;
        }
```

(9) 매수 페이지 ('/transactions/buy')

매수 페이지에서는 실제 해당 종목의 매수를 처리하기 위한 기능을 제공한다. 우선 해당 주식 종목의 코드와 종목명을 보여주고, 5단계 호가창을 보여준다. 5단계 호가창을 보여주기 위해 '/transactions/buy'을 처리하는 백엔드 코드에서 높은 가격부터 상위 5개의 매도 주문과 상위 5개의 매수 주문을 읽는다. 위 정보를 읽는 sql query는 다음과 같다. 이때 호가창의 경우 미체결된 주문에 대해서 수행해야 하기 때문에 left_quantity > 0 인 조건을 추가한다.

```
// 매도 주문 (높은 가격부터 상위 5 개)
      const [sellOrders] = await sql_connection.promise().query(
         `SELECT sum(left_quantity) as sell_quantity, price as sell_price
          FROM Transaction
          WHERE stock_id = ? AND type = 'SELL' AND left_quantity > 0
          GROUP BY price
          ORDER BY price DESC
          LIMIT 5`,
         [stock_id]
      );

      // 매수 주문 (높은 가격부터 상위 5 개)
      const [buyOrders] = await sql_connection.promise().query(
         `SELECT sum(left_quantity) as buy_quantity, price as buy_price
          FROM Transaction
          WHERE stock_id = ? AND type = 'BUY' AND left_quantity > 0
          GROUP BY price
          ORDER BY price DESC
          LIMIT 5`,
         [stock_id]
      );
```

이후 잔고를 띄워주기 위해, Account 테이블에서 해당 account_id의 예수금을 읽어오고, 보유수량을 읽어 오기 위해 Owning_stock에서 해당 account_id와 해당 stock_id에 해당하는 quantity 값을 읽어온다. 위 sql 코드는 다음과 같다.

```
// Account 정보에서 잔고 조회
      const [[accountInfo]] = await sql_connection.promise().query(
         `SELECT balance
          FROM Account
          WHERE account_id = ?`,
         [account_id]
      );
// 해당 종목의 보유 수량 조회
      const [holdingResult] = await sql_connection.promise().query(
         `SELECT quantity
          FROM Owning_stock
          WHERE account_id = ? AND stock_id = ?`,
         [account_id, stock_id]
```

```
        );
```

해당 보유 수량이 없다면 0으로 설정하는 코드가 뒤이어 실행된다.

이후, 위 정보를 넘겨서 stockBuy.ejs라는 페이지를 렌더링한다. 해당 페이지는 이렇게 받은 정보로 다음과 같이 5단계 호가창을 보여준다.

```
<!-- 5단계 호가창: 매도 호가와 매수 호가를 각각 높은 가격 순으로 표시 -->
          <% for (let i = 0; i < 5; i++) { %>
              <tr>
                  <% const index = 5 - sellOrders.length; %>

                  <% if (i < index) { %> <!-- 빈 칸을 먼저 출력 -->
                      <td>-</td>
                      <td>-</td>
                      <td></td>
                      <td></td>
                  <% } else { %> <!-- 데이터가 있는 경우만 sellOrders 접근 -->
                      <td><%= sellOrders[i - index] ? sellOrders[i -
index].sell_quantity : '-' %></td>
                      <td><%= sellOrders[i - index] ? sellOrders[i -
index].sell_price : '-' %></td>
                      <td></td>
                      <td></td>
                  <% } %>
              </tr>
          <% } %>

          <% for (let j = 0; j < 5; j++) { %>
              <tr>
                  <td></td>
                  <td></td>
                  <td><%= buyOrders[j] ? buyOrders[j].buy_price : '-' %></td>
                  <td><%= buyOrders[j] ? buyOrders[j].buy_quantity : '-' %></td>
              </tr>
          <% } %>
```

해당 페이지에는 밑부분에 시장가 매수와 지정가 매수 기능을 제공하는데, 시장가 매수의 경우 수량을 입력하고 매수 버튼을 누르면 '/transactions/market-buy'라는 url의 백엔드 코드가 실행되고 우선 매도 주문을 시장가로 매수하기 위해 해당 정보를 price의 오름차순으로 가져온다. 만약 값이 같다면 시간이 오래된 순으로 가져온다.

```
// 매도 주문을 시장가로 매수
      const [sellOrders] = await sql_connection.promise().query(
          `SELECT price, left_quantity, transaction_id, account_id as
seller_account_id
           FROM Transaction
           WHERE stock_id = ? and type = 'SELL' and left_quantity > 0
           ORDER BY price, date, time`,
          [stock_id]
```

```
        );
```

이후 remainingQuantity = quantity;로 설정하고, 예수금이 충분한지 확인하기 위해 sellOrders의 주문을 차례로 돌며 다 매수 했을 때의 totalCost를 계산하고 이것이 예수금보다 크다면 해당 매수 주문을 거절한다. 만약 괜찮다면, 계좌 잔액을 총비용으로 업데이트한다.

```
// 계좌 잔액 업데이트
        await sql_connection.promise().query(
            `UPDATE Account
             SET balance = balance - ?
             WHERE account_id = ?`,
            [totalCost, account_id]
        );
```

이후 실제 매수 처리를 위해 매도 주문을 낮은 가격부터 훑으면서 매도 주문을 업데이트 한다. 이때 거래수량은 remainingQuantity와 해당 매도 주문의 left_quantity 중 작은 값을 사용한다.

```
// 매도 주문 업데이트
            await sql_connection.promise().query(
                `UPDATE Transaction
                 SET left_quantity = left_quantity - ?
                 WHERE transaction_id = ?`,
                [tradeQuantity, transaction_id]
            );
```

이후 매도자의 보유 주식 수를 차감한다. 이때 매도자의 보유 주식 수가 0이 된다면 Owning_stock 테이블에서 해당 매도자의 해당 주식에 대한 튜플을 삭제한다. 또 매도자의 예수금에 cost만큼 올려준다.

```
// 매도자 보유 주식 수 업데이트
            await sql_connection.promise().query(
                `UPDATE Owning_stock
                 SET quantity = quantity - ?
                 WHERE account_id = ? and stock_id = ?`,
                 [tradeQuantity, seller_account_id, stock_id]
            );

            // 매도자의 보유 주식 수가 0 이 되는 경우
            const [[{ updatedQuantity }]] = await sql_connection.promise().query(
                `SELECT coalesce(quantity, 0) as updatedQuantity
                 FROM Owning_stock
                 WHERE account_id = ? and stock_id = ?`,
                [seller_account_id, stock_id]
            );

            if (updatedQuantity <= 0) {
                await sql_connection.promise().query(
                    `DELETE FROM Owning_stock
                     WHERE account_id = ? and stock_id = ?`,
                     [seller_account_id, stock_id]
                );
            }
```

```
        // 매도자 예수금 업데이트
        await sql_connection.promise().query(
            `UPDATE Account
             SET balance = balance + ?
             WHERE account_id = ?`,
             [cost, seller_account_id]
        );
```

이후엔 매수자 처리를 해줘야 하는데 우선 매수자의 보유 주식 정보를 업데이트 하기 위해 해당 주식 종목에 대한 값을 튜플을 읽고, 만약 있다면 해당 수량과 평단가를 업데이트 해준다. 평단가는 기존 수량 * 평단가에 새로운 cost를 더한 뒤 바뀐 수량으로 나눠 계산한다. 만약 없다면, 해당 종목에 대해 새로운 항목을 만들어준다.

```
// 매수자 보유 주식 정보 업데이트
        const [accountStockInfo] = await sql_connection.promise().query(
            `SELECT *
             FROM Owning_stock
             WHERE account_id = ? and stock_id = ?`,
             [account_id, stock_id]
        );

        // 이미 해당 종목 있는 경우 수량, 평단가 update
        if (accountStockInfo.length > 0) {
            let newBuyingPrice = accountStockInfo[0].buying_price *
accountStockInfo[0].quantity;
            newBuyingPrice += cost;
            newBuyingPrice = parseFloat((newBuyingPrice /
(accountStockInfo[0].quantity + tradeQuantity)).toFixed(2));

            await sql_connection.promise().query(
                `UPDATE Owning_stock
                 SET quantity = quantity + ?,
                     buying_price = ?
                 WHERE account_id = ? and stock_id = ?`,
                 [tradeQuantity, newBuyingPrice, account_id, stock_id]
            );
        }
        else {  // 없는 경우 새로 항목 만들기
            await sql_connection.promise().query(
                `INSERT INTO Owning_stock (account_id, stock_id, quantity,
buying_price) VALUES
                 (?, ?, ?, ?)`,
                 [account_id, stock_id, tradeQuantity, price]
            );
        }
```

이후 해당 수량만큼의 주문 record를 현재 시간으로 해서 추가해준다.

```
// 주문 record 추가
        await sql_connection.promise().query(
```

```
            `INSERT INTO Transaction (type, account_id, stock_id, price, quantity,
left_quantity, date, time) VALUES
             ('BUY', ?, ?, ?, ?, ?, CURDATE(), CURTIME())`,
            [account_id, stock_id, price, tradeQuantity, 0]
        );
```

이후 더이상 remainQuantity가 없다면 반복문을 빠져 나온다. 만약 반복문이 끝났는데도 매수 잔량이 남은 경우에는, 매도 주문이 있다면 매도 주문의 가장 낮은 값으로 설정하고, 그게 아니라면 가장 높은 매수 호가로 걸어둔다.

```
// 시장가 매수 잔량이 남은 경우
      if (remainingQuantity !== 0) {
         // 매도 주문이 없는 경우, 현재 가장 높은 매수 호가로 걸어두기
         if (sellOrders.length === 0) {
            const [[{ buyPrice }]] = await sql_connection.promise().query(
               `SELECT price as buyPrice
                FROM Transaction
                WHERE stock_id = ? and type = 'BUY' and left_quantity > 0
                ORDER BY price desc
                LIMIT 1`,
                [stock_id]
            );

            // 주문 record 추가
            await sql_connection.promise().query(
               `INSERT INTO Transaction (type, account_id, stock_id, price,
quantity, left_quantity, date, time) VALUES
               ('BUY', ?, ?, ?, ?, ?, CURDATE(), CURTIME())`,
               [account_id, stock_id, buyPrice, remainingQuantity,
remainingQuantity]
            );
         }
         // 주문 record 추가
         await sql_connection.promise().query(
            `INSERT INTO Transaction (type, account_id, stock_id, price, quantity,
left_quantity, date, time) VALUES
             ('BUY', ?, ?, ?, ?, ?, CURDATE(), CURTIME())`,
            [account_id, stock_id, sellOrders[-1].price, remainingQuantity,
remainingQuantity]
         );
      }
```

지정가 매수의 경우는 시장가 매수와 비슷하다. 다만, 가격을 지정하기 때문에 계좌 잔액을 먼저 업데이트 하고, 지정가 매수 주문 record를 추가한다.

```
// 계좌 잔액 업데이트
      await sql_connection.promise().query(
         `UPDATE Account
          SET balance = balance - ?
          WHERE account_id = ?`,
         [totalCost, account_id]
      );
```

```
      // 지정가 매수 주문 record 추가
      const [result] = await sql_connection.promise().query(
         `INSERT INTO Transaction (type, account_id, stock_id, price, quantity,
left_quantity, date, time) VALUES
          ('BUY', ?, ?, ?, ?, ?, CURDATE(), CURTIME())`,
         [account_id, stock_id, price, quantity, quantity]
      );
```

이후 과정은 동일하게 매도 주문을 찾는데 이때 조건에 price를 입력한 지정가로 설정해서 매도 주문 데이터 쿼리를 만든다.

```
/ 거래 체결을 위해 매도 주문 검색
      const [sellOrders] = await sql_connection.promise().query(
         `SELECT left_quantity, transaction_id, account_id as seller_account_id
          FROM Transaction
          WHERE stock_id = ? and type = 'SELL' and left_quantity > 0 and price = ?
          ORDER BY date, time`,
         [stock_id, price]
      );
```

이후 동일하게 remainingQuantity를 줄여가며 해당 매도 주문과 상쇄하며 거래를 체결시킨다. 이 과정에서 매도 주문 업데이트, 매도자 보유 주식 수 업데이트, 매도자 예수금 업데이트, 매수 주문 잔여 수량을 줄이고, 매수자 보유 주식 정보를 업데이트 하는 과정은 시장가 매수와 동일하다.

(10) 매도 페이지 ('/transactions/sell')

마지막으로 매도 페이지의 경우, 위의 5단계 호가창은 매수 페이지와 동일하다. 시장가 매도의 경우에는 거래 가능 수량을 의미하는 tradable_quantity와 주문의 quantity를 비교하고 작거나 같은 경우에만 매도 주문을 허용한다. 여기서는 매수와 반대로 매수 호가를 높은 가격부터 조회한다.

```
// 매수 호가 조회 (높은 가격부터)
      const [buyOrders] = await sql_connection.promise().query(
         `SELECT price, left_quantity, transaction_id, account_id as
buyer_account_id
          FROM Transaction
          WHERE stock_id = ? and type = 'BUY' and left_quantity > 0
          ORDER BY price desc, date, time`,
         [stock_id]
      );
```

마찬가지로 remainingQuantity를 설정하고, 매수 주문을 하나씩 반복문으로 훑으며 시장가 매수와 동일하게 거래수량과 Cost를 계산한다. 다만 처리 로직이 조금 다른데, 우선 매수 주문을 업데이트한다. Transaction 테이블의 left_quantity를 거래 수량만큼 줄이고, 매수자 보유 주식 정보 업데이트를 위해 매수자의 보유 주식 정보를 Owning_stock에서 찾는다. 이후 해당 종목이 있는 경우 수량, 평단가를 업데이트한다. 업데이트 로직은 매수와 같다.

```
// 매수 주문 업데이트
         await sql_connection.promise().query(
```

```
            `UPDATE Transaction
             SET left_quantity = left_quantity - ?
             WHERE transaction_id = ?`,
            [tradeQuantity, transaction_id]
        );

        // 매수자 보유 주식 정보 업데이트
        const [accountStockInfo] = await sql_connection.promise().query(
            `SELECT *
             FROM Owning_stock
             WHERE account_id = ? and stock_id = ?`,
             [buyer_account_id, stock_id]
        );

        // 이미 해당 종목 있는 경우 수량, 평단가, 수량 update
        if (accountStockInfo.length > 0) {
            let newBuyingPrice = accountStockInfo[0].buying_price *
accountStockInfo[0].quantity;
            newBuyingPrice += cost;
            newBuyingPrice = parseFloat((newBuyingPrice /
(accountStockInfo[0].quantity + tradeQuantity)).toFixed(2));

            await sql_connection.promise().query(
                `UPDATE Owning_stock
                 SET quantity = quantity + ?,
                     buying_price = ?
                 WHERE account_id = ? and stock_id = ?`,
                 [tradeQuantity, newBuyingPrice, buyer_account_id, stock_id]
            );
        }
        else {  // 없는 경우 새로 항목 만들기
            await sql_connection.promise().query(
                `INSERT INTO Owning_stock (account_id, stock_id, quantity,
buying_price) VALUES
                 (?, ?, ?, ?)`,
                 [buyer_account_id, stock_id, tradeQuantity, price]
            );
        }
```

이후, 매도자 보유 주식 수를 차감하고, 위와 똑같이 보유 주식 수가 0이 되면 해당 종목을 보유 목록에서 삭제한다.

```
// 매도자 보유 주식 수 업데이트
        await sql_connection.promise().query(
            `UPDATE Owning_stock
             SET quantity = quantity - ?
             WHERE account_id = ? and stock_id = ?`,
             [tradeQuantity, account_id, stock_id]
        );

         // 매도자의 보유 주식 수가 0 이 되는 경우, 해당 종목 보유 삭제
```

```
            const [[{ updatedQuantity }]] = await sql_connection.promise().query(
                `SELECT quantity as updatedQuantity
                 FROM Owning_stock
                 WHERE account_id = ? and stock_id = ?`,
                [account_id, stock_id]
            );

            if (updatedQuantity <= 0) {
                await sql_connection.promise().query(
                    `DELETE FROM Owning_stock
                     WHERE account_id = ? and stock_id = ?`,
                     [account_id, stock_id]
                );
            }
```

이후 매도자 예수금을 cost 만큼 더해주고, 주문 record를 추가한다. 여기서도 remainingQuantity가 0이 되면 반복문을 나온다. 시장가 매도 잔량이 남은 경우 똑같이 매수 주문이 있다면 가장 높았던 마지막 매수 주문 가격으로 걸어두고, 없다면 가장 낮은 매도 호가로 걸어둔다.

```
// 시장가 매도 잔량이 남은 경우 (걸어 놓기)
        if (remainingQuantity !== 0) {
            // 주문 record 추가 (매수 주문이 없는 경우, 현재 가장 낮은 매도 호가로 걸어두기)
            if (buyOrders.length === 0) {
                const [[{ sellPrice }]] = await sql_connection.promise().query(
                    `SELECT price as sellPrice
                     FROM Transaction
                     WHERE stock_id = ? and type = 'SELL' and left_quantity > 0
                     ORDER BY price
                     LIMIT 1`,
                     [stock_id]
                );

                await sql_connection.promise().query(
                    `INSERT INTO Transaction (type, account_id, stock_id, price,
quantity, left_quantity, date, time) VALUES
                     ('SELL', ?, ?, ?, ?, ?, CURDATE(), CURTIME())`,
                     [account_id, stock_id, sellPrice, remainingQuantity,
remainingQuantity]
                );
            }
            else {
                await sql_connection.promise().query(
                    `INSERT INTO Transaction (type, account_id, stock_id, price,
quantity, left_quantity, date, time) VALUES
                     ('SELL', ?, ?, ?, ?, ?, CURDATE(), CURTIME())`,
                     [account_id, stock_id, buyOrders[buyOrders.length - 1].price,
remainingQuantity, remainingQuantity]
                );
            }
        }
```

지정가 매도는 시장가 매도와 비슷하지만 다른 점은 지정가 매도 주문 record를 먼저 추가한다는

점이다.

```
// 지정가 매도 주문 record 추가
      const [result] = await sql_connection.promise().query(
         `INSERT INTO Transaction (type, account_id, stock_id, price, quantity,
left_quantity, date, time) VALUES
             ('SELL', ?, ?, ?, ?, CURDATE(), CURTIME())`,
             [account_id, stock_id, price, quantity, quantity]
      );
```

이후 과정은 매수 주문을 훑으면서 매수 주문을 업데이트,. 매수자 보유 주식 정보 업데이트, 매도자 보유 주식 수 업데이트, 매도자 예수금 업데이트, 매도 잔여 수량 줄이기의 과정으로 동일하다. 다른 점은 price를 지정한다는 점이다.

3. Trouble Shooting

이번 과제를 하면서 수많은 error를 만났다. 특히 어떤 변수가 undefined라는 말을 참 많이 본 것 같다. 하지만, 오랜 시간을 들인 끝에 명세에서 언급한 것들을 다 완성할 수 있었다.

시연 영상 URL : https://www.youtube.com/watch?v=I-0i-BFEvyA