

WIKI

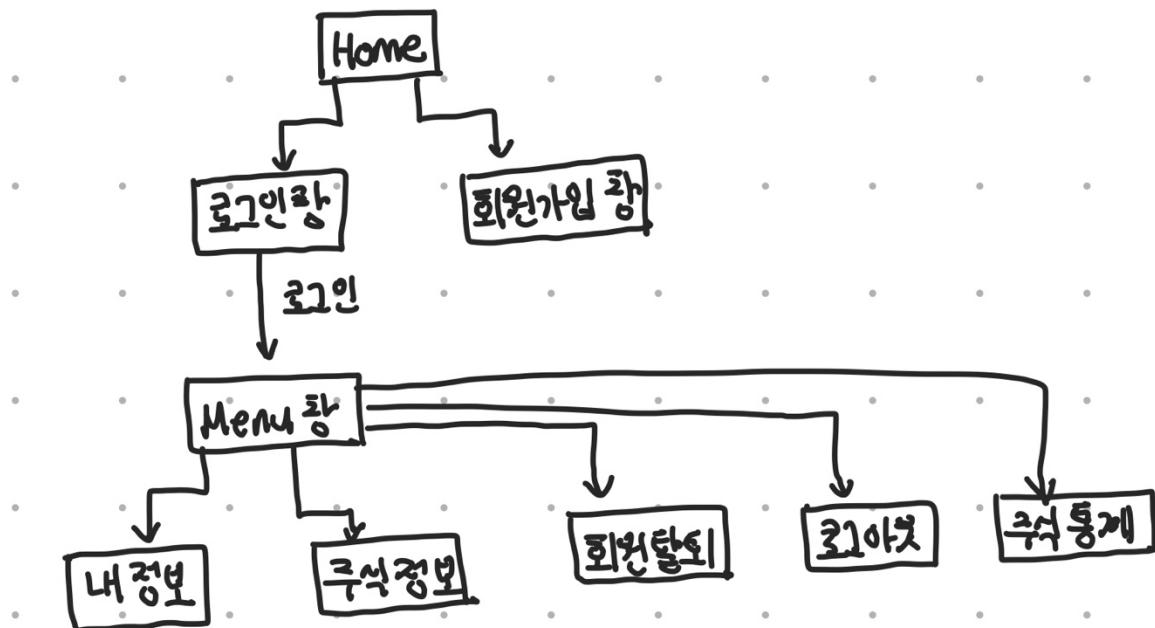
이름: 문준영

학과: 컴퓨터소프트웨어학부

학번: 2022006135

Design

본 프로젝트를 시작하기 앞서, 먼저 어떻게 웹 페이지를 구성할 것인지를 고민해보았다. 페이지는 몇 개로 이루어져 있으며 어떠한 기능을 어디서 구현할지를 생각했다.



이것이 내가 생각한 웹 사이트의 구조인데 이를 설명하면 다음과 같다. 먼저, 처음에 사이트를 접근하면 Home 화면이 등장한다. Home 화면에서는 로그인 창과 회원가입 창으로 이동할 수 있다. 로그인이 완료되면 Menu 창에 들어가게 된다. Menu 창에서는 내 정보, 주식정보, 회원탈퇴, 로그아웃, 그리고 주식 통계 창으로 이동 할 수 있다., 단순한 구조로 짜보아도 상당히 많은 수의 페이지가 필요하기에 app.js에 모든 것은 만들지 않고

```
JS home.js ×
db_application > routes > JS home.js > ...
1 const express = ...
2 const router = express.Router();
```

express의 router를 사용하기로 하였다. 다음은 home.js에서의 router 사용 예시이다. Router를 선언해주고

```
module.exports = router;    이를 exports 해준다.
```

app.js에서 homeRouter를 선언해주고

```
const homeRouter = require('./routes/home');
```

app.use를 통해 '/' 경로에 있는 모든 처리는 homeRouter를 통해 한다는 것을 선언한다.

```
app.use('/', homeRouter);
```

또한, view에 모든 ejs파일을 넣어 파일을 잘 정돈하였다.

이를 통해 구조적으로 웹 코딩을 할 수 있었다.

```
app.set('view engine', 'ejs');
app.set('views', './views');

app.use('/', homeRouter);
app.use('/login', loginRouter);
app.use('/signup', signupRouter);
app.use('/menu', menuRouter);
app.use('/userInfo', userInfoRouter);
app.use('/deposit', depositRouter);
app.use('/withdraw', withdrawRouter);
app.use('/stockInfo', stockInfoRouter);
app.use('/signout', signoutRouter);
app.use('/buy', buyRouter);
app.use('/sell', sellRouter);
app.use('/holdingStock', holdingStockRouter);
app.use('/userTradingLog', userTradingLogRouter);
app.use('/stockTradingLog', stockTradingLogRouter);
app.use('/orderBook', orderBookRouter);
app.use('/tradingVolume', tradingVolumeRouter);
```

(app.js 파일에서의 Router 활용)

각 기능에 대해 웹 페이지가 어떻게 어떤 기능으로 구성되었는지를 살펴보도록 하겠다.

● Home

처음 페이지에 접근했을 때 볼 수 있는 화면이다. 로그인과 회원가입 버튼이 있다.

● 회원가입

회원가입을 위해 아이디, 비밀번호, 사용자 이름, 나이, 예수금을 입력으로 받는데. 만약 같은 아이디가 존재하면 회원가입이 실패되도록 하였다.

- 로그인

localhost:3000/login

로그인

아이디 비밀번호

로그인

로그인 페이지에서는 아이디와 비밀번호를 입력으로 받는데. 해당 아이디에 대해 비밀번호가 맞다면 로그인이 성공하고 Menu 페이지로 이동하게 된다.

- Menu

localhost:3000/login

Menu

내 정보 주식 정보 회원 탈퇴 로그아웃 거래량 통계

Menu 페이지에는 5개의 버튼이 있다. 각각 내 정보, 주식 정보, 회원 탈퇴, 로그아웃, 거래량 통계를 할 수 있는 곳으로 이동한다.

● 내 정보

localhost:3000/userInfo

문준영
6789500

입금 출금

보유 주식 보기
주식 거래 내역 보기

내 정보 페이지에서는 현재 사용자의 이름과 예수금을 볼 수 있다. 예수금 옆에는 '입금'과 '출금'을 할 수 있는 버튼이 있다.

입금

입금액 입금

입금 페이지에서는 입금액을 입력받고 이를 통해 예수금을 증가시킨다.

출금

출금액 출금

출금을 하는 경우에는 반대로 출금액을 입력으로 받아 그만큼 예수금을 감소시킨다.

내 정보에서 보유 주식을 보기 위해 '보유 주식 보기'를 누르면 보유 주식을 볼 수 있다.

← → ⌂ ⓘ localhost:3000/holdingStock

▤ Baekjoon Online J... ⓘ 한양대학교 Ims ☐ 컴퓨터비전

검색

종목명 매수가 현재가 보유수량 거래가능수량 평가손익 등락률 (%)

수용전자	57380	57200	100	80	-18000	-0.3137
정현차	206300	206300	10	10	0	0.0000
OS	76500	76500	10	10	0	0.0000

보유주식에 대한 정보와 해당 주식에 대한 평가손익과 등락률까지 표시하였다. 또한 종목명으로 부분일치 검색이 가능하다.

내 정보에서 주식 거래 내역을 보기 위해서는 '주식 거래 내역 보기'를 눌러 현재 사용자의 주식 거래 내역을 볼 수 있다.

← → ⌂ ⓘ localhost:3000/userTradingLog

▤ Baekjoon Online J... ⓘ 한양대학교 Ims ☐ 컴퓨터비전

검색

종목명 매수가(-)/매도가(+) 거래 수량

거래 시각

수용전자	57200	10	Wed Nov 06 2024 21:06:53 GMT+0900 (대한민국 표준시)
수용전자	-57200	10	Wed Nov 06 2024 21:06:53 GMT+0900 (대한민국 표준시)
정현차	-206300	10	Wed Nov 06 2024 21:08:28 GMT+0900 (대한민국 표준시)
OS	-76500	10	Wed Nov 06 2024 21:10:47 GMT+0900 (대한민국 표준시)

해당 주식에 대한 거래가 매수인 경우에는 음수로 매수가를 표현하였고 매도가 인 경우에는 양수로 매도가를 표현하였다. 이 역시 종목명으로 부분 일치 검색이 가능하다.

● 주식 정보.

← → ⌂ ⓘ localhost:3000/stockInfo

▤ Baekjoon Online J... ⓘ 한양대학교 Ims ☐ 컴퓨터비전

검색

종목 ID 종목명 현재가 전일 종가 거래량 전일 대비 주가 등락률 매수 매도 주식 거래 내역

1	수용전자	57200	57600	10	-0.6944	매수	매도	주식 거래 내역	호가창 보기
2	정현차	206300	215000	10	-4.0465	매수	매도	주식 거래 내역	호가창 보기
3	OS	76500	76700	10	-0.2608	매수	매도	주식 거래 내역	호가창 보기
4	DB	176300	176500	0	-0.1133	매수	매도	주식 거래 내역	호가창 보기
5	언규군사물자	10000	10300	0	-2.9126	매수	매도	주식 거래 내역	호가창 보기

Menu 페이지에서 주식 정보를 클릭하면 모든 주식에 대한 정보가 나오게 된다. 여기서도 종목명으로 부분일치 검색이 가능하다.

해당 주식을 매수하기 위해서는 '매수'버튼을 클릭하면 된다.

← → ⌛ ⓘ localhost:3000/buy

▤ Baekjoon Online J... ⏪ 한양대학교 Ims ⏮ 컴퓨터비전

매수

수량 금액 시장가

매수

매수창에서는 수량과 금액을 입력으로 받아 매수 주문을 넣는다. 옆에 있는 시장가 버튼을 누르면 시장가가 금액란에 들어가게 된다.

← → ⌛ ⓘ localhost:3000/sell

▤ Baekjoon Online J... ⏪ 한양대학교 Ims ⏮ 컴퓨터비전

매도

수량 금액 시장가

매도

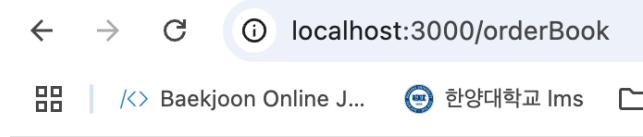
매도의 경우에도 매수와 마찬가지로 수량과 금액을 입력으로 받는다. 옆에 시장가를 누르면 시장가가 금액란에 들어가게 된다. 이를 통해 해당 주식에 대한 매도 주문을 걸 수 있다.

← → ⌛ ⓘ localhost:3000/stockTradingLog

▤ Baekjoon Online J... ⏪ 한양대학교 Ims ⏮ 컴퓨터비전

매수자 ID	매도자 ID	거래 가격	거래 수량	거래 시각
joy999871	joy999871	57200	10	Wed Nov 06 2024 21:06:53 GMT+0900 (대한민국 표준시)

해당 종목에 대한 주식 거래 내역을 보려면 해당 종목의 '주식 거래 내역'을 클릭하면 된다. 이를 클릭하면 종목에 대한 거래 내역이 나온다.



OS

76500

매도수량 호가 매수수량

- 77000 -
- 76900 -
- 76800 -
- 76700 -
- 76600 -
- 76500 5
- 76400 -
- 76300 -
- 76200 -
- 76100 -

해당 주식에 대한 호가창을 보기 위해서는 해당 주식 옆에 '호가창 보기' 버튼을 누르면 된다. 호가창은 현재가를 기준으로 5단계 호가창이 구성되어 있다. 위에 표시되는 것은 종목 명과 현재가이다. 왼쪽 사진에서는 76500에 매수주문이 5주 걸려있는 것을 볼 수 있다.

● 회원 탈퇴

회원 탈퇴

아이디	<input type="text"/>	비밀번호	<input type="password"/>
<input type="button" value="회원 탈퇴"/>			

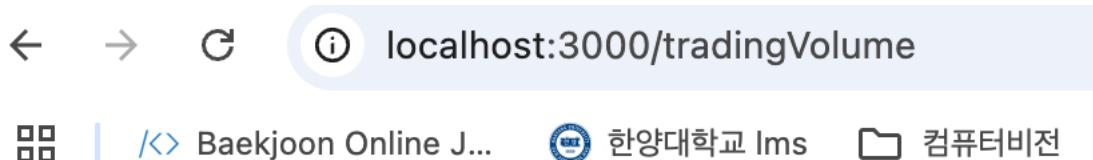
회원 탈퇴에서는 아이디와 비밀번호를 입력해 현재 사용자의 아이디, 비밀번호와 입력

한 값이 일치하면 회원 탈퇴가 되도록 구현 할 것이다.

● 로그아웃

로그아웃은 별도의 창을 만든 것이 아닌 현재 사용자를 지우고 (session을 지우고) home 화면으로 돌아가게 한다.

● 주식 통계

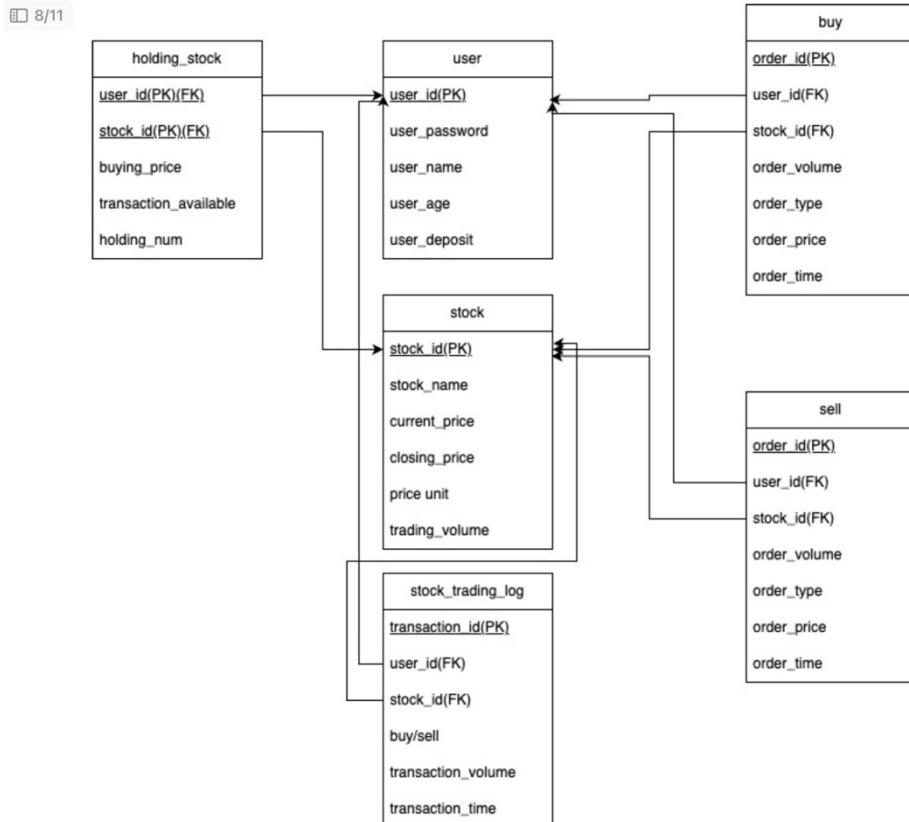


거래량 순위	종목 ID	종목명	현재가	거래량
1	1	수용전자	57200	10
2	2	정현차	206300	10
3	3	OS	76500	10
4	4	DB	176300	0
5	5	언규군사물자	10000	0

주식 통계로는 거래량 순위를 볼 것이다.

Implementation

본격적인 구현에 대해 설명해 보겠다. 각 페이지 별로 구현한 기능에 대해서 상세하게 설명해 보도록 하겠다. 주로 SQL 문을 통해 필요한 페이지에 대한 설명을 주로 하겠다. SQL 문의 이해의 편의를 위해 relational model을 첨부하겠다.



● Signup (회원가입)

먼저 회원가입 페이지부터 보자. 먼저 ejs파일을 통해 user_id (아이디), user_password (비밀번호), user_name (사용자 이름), user_age (사용자 나이), user_deposit (예수금)을 받는다.

```

<form method="post">
  <table>
    <tr>
      <td><label>아이디</label></td>
      <td>
        <label>
          <input type="text" name="userID">
        </label>
      </td>
      <td><label>비밀번호</label></td>
      <td>
        <label>
          <input type="text" name="userPassword">
        </label>
      </td>
      <td><label>사용자 이름</label></td>
      <td>
        <label>
          <input type="text" name="userName">
        </label>
      </td>
    </tr>
  </table>
</form>

```

이에 대한 ejs파일인데 입력을 받아 post 형식으로 회원가입 페이지에 전달한다.

```

const {userID, userPassword, userName, userAge, userDeposit} = req.body;
if(userAge < 0){
  return res.render('signupPage', {signupAge : 'negative'});
}
else if(userDeposit < 0){
  return res.render('signupPage', {signupDeposit : 'negative'});
}
else if(userDeposit > MAX_DEPOSIT){
  return res.render('signupPage', {signupDeposit : 'overflow'});
}

```

Post로 전달된 정보를 서버에서 받아 입력된 값이 오류가 있는지 검사한다. (나이가 음수인지, 예수금이 음수인지, 예수금이 INT의 범위를 넘는지) 만약 오류가 있다면 회원가입을 실패하게 한다.

```

db.query("SELECT * FROM user WHERE user_id = ?", [userID], (error, results, fields) => {
  if (error) throw error;
  if (results.length > 0){
    res.render('signupPage', {ExistingID : "True"});
  }
  else{
    db.query("INSERT INTO user(user_id, user_password, user_name, user_age, user_deposit) values (?, ?, ?, ?, ?, ?)",
      [userID, userPassword, userName, userAge, userDeposit], (error, results, fields) => {
        if (error) throw error;
        else{
          res.render('home', {signup : "success"});
        }
      });
  }
});

```

오류가 없음을 확인한 후 입력으로 들어온 userID가 이미 존재하는지를 확인한다. 존재하지 않으면 새로운 user로 추가해주고 회원가입을 완료하게 한다.

● Login (로그인)

Login 페이지 역시 signup 페이지와 마찬가지로 userID와 userPassword를 post 형식으로 받는다.

```

9  router.post('/', (req, res) => {
10    const {userID, userPassword} = req.body;
11    db.query("SELECT * FROM user WHERE user_id = ? AND user_password = ?", [userID, userPassword],
12    [error, results, fields] => {
13      if (error) throw error;
14      if(results.length > 0){
15        req.session.userID = userID;
16        req.session.userPassword = userPassword;
17        res.render('menuPage');
18      }
19      else{
20        res.render("loginPage", {loginResult: 'fail'});
21      }
22    })
23  )

```

User relation에서 입력으로 받은 userID와 userPassword와 값이 모두 일치하는 tuple을 찾는다. 만약 그러한 tuple이 존재하지 않으면 로그인이 실패하게 된다. 그리고 해당 user의 정보를 로그인 상태에서 기억하고 있어야 하므로 session을 사용해 userID를 저장한다. 로그인 성공 후 Menu 페이지로 이동한다.

Menu 페이지에서는 앞서 말했다시피 다섯 개의 버튼이 존재한다. 각각은 "내 정보", "주식 정보", "회원 탈퇴", "로그아웃", "거래량 통계"를 볼 수 있는 페이지로 이동시킨다. 각각에 대해 구현 방식을 설명하겠다.

● UserInfo Page (내 정보)

```

router.get('/', (req, res) => {
  db.query('SELECT * FROM user WHERE user_id = ?', [req.session.userID], (error, results, fields) => {
    if(error) throw error;
    if(results.length > 0){
      const userName = results[0].user_name;
      const userDeposit = results[0].user_deposit;
      res.render('userInfoPage', {userName : userName, userDeposit : userDeposit});
    }
    else{
      res.send('사용자를 찾을 수 없습니다.');
    }
  });
});

```

UserInfo 페이지에 앞서 먼저 session을 이용해 user relation으로 부터 user_name, user_deposit을 가져와 이를 ejs에 넘겨준다. 때문에 웹 페이지에서는 현재 사용자의 이름과 예수금을 볼 수 있다.

```

<tr>
  <td><label>이름</label></td>
  <td><%= userName %></td>
</tr>
<tr>
  <td><label>예수금</label></td>
  <td><%= userDeposit %></td>
  <td><button id='deposit'>입금</button></td>
  <td><button id='withdraw'>출금</button></td>
</tr>

```

예수금 옆에 입금과 출금 버튼이 있다. 이를 클릭하면 입금과 출금 페이지로 이동한다.

● Deposit (입금)

앞서 본 입금 페이지대로 post 형식으로 금액을 입력으로 받는다.

```

router.post('/', (req, res) => {
  const {depositAmount} = req.body;
  const userID = req.session.userID;
  const deposit = parseInt(depositAmount);
  console.log(depositAmount);
  if(deposit < 0){
    return res.render('depositPage', {negativeValue : "negative"});
  }
  db.query("SELECT * FROM user WHERE user_id = ?", [userID], (error, results, fields) => {
    if(error) throw error;
    if(results.length > 0){
      const currentDeposit = parseInt(results[0].user_deposit);
      const newDeposit = currentDeposit + deposit;
      if(newDeposit > MAX_DEPOSIT){
        return res.render('depositPage', {negativeValue : 'overflow'});
      }
      db.query("UPDATE user SET user_deposit = ? WHERE user_id = ?", [newDeposit, userID], (error, results, fields) => {
        if(error) throw error;
        res.render('menuPage', {Deposit : 'success'});
      });
    }
  });
}

```

먼저, 입력받은 금액이 음수면 입금 실패하여 다시 입금 페이지로 돌아가게 된다. Session을 통해 사용자를 찾고 입금 후 새로운 예수금을 Update SQL 쿼리를 통해 넣어준다. (이 과정에서 새로운 예수금이 INT의 범위를 넘으면 안되니 이를 검사해준다.)

● Withdraw (출금)

```

router.post('/', (req, res) => {
  const {withdrawAmount} = req.body;
  const userID = req.session.userID;
  const withdraw = parseInt(withdrawAmount);
  if(withdraw < 0){
    return res.render('withdrawPage', {negativeValue : "negative"});
  }
  db.query("SELECT * FROM user WHERE user_id = ?", [userID], (error, results, fields) => {
    if(error) throw error;
    if(results.length > 0){
      const currentDeposit = parseInt(results[0].user_deposit);
      const newDeposit = currentDeposit - withdraw;
      if(newDeposit < 0){
        return res.render('depositPage', {negativeValue : 'overflow'});
      }
      db.query("UPDATE user SET user_deposit = ? WHERE user_id = ?", [newDeposit, userID], (error, results, fields) => {
        if(error) throw error;
        res.render('menuPage', {Withdraw : 'success'});
      });
    }
  });
}

```

출금의 경우도 입금과 마찬가지로 동작한다. 먼저, 들어온 금액이 음수면 오류를 발생시키고 이번에는 새로운 예수금을 현재 예수금 - 출금액 형식으로 정의하여 예수금을 감소시킨다.

● Holding Stock (보유주식)

현재 사용자의 보유주식을 보기 위해서는 userInfo 페이지에서 '보유 주식 보기' 버튼을 클릭하면 된다. 처음 보유 주식 페이지에 들어갈 때는 현재 사용자가 보유한 모든 주식

을 볼 수 있도록 하였다.

```
var query = `SELECT S.stock_name, H.buying_price, S.current_price,
H.holding_num, H.transaction_available, (((S.current_price)- (H.buying_price)) * (H.holding_num)) as Sonik,
(((S.current_price) - (H.buying_price)) / H.buying_price) * 100) as up_down
FROM holding_stock as H JOIN stock as S ON H.stock_id = S.stock_id
WHERE H.user_id = ?`
```

이 쿼리를 보면 먼저 보유한 주식의 종목명과 현재가가 필요하기에 holding_stock relation과 stock relation을 stock_id를 통해 join 하였다. 또한, 현재 user에 대한 보유 주식만을 볼 수 있도록 하기 위해 WHERE 문에서 user_id의 일치 여부를 검사하였다. 추가적으로, 평가손익과, 매수가 대비 등락률을 알 수 있도록 평가손익과 등락률 역시 동시에 볼 수 있도록 하였다.

'보유 주식' 페이지에서는 앞서 봤듯이 종목명으로 부분 일치 검색을 하는 기능이 있다. Post 형식으로 종목명을 입력으로 받으면 이 쿼리를 통해 해당 종목을 찾는다.

```
var query2 = `SELECT S.stock_name, H.buying_price, S.current_price, H.holding_num, H.transaction_available,
((S.current_price)- (H.buying_price)) * (H.holding_num)) as Sonik,
(((S.current_price) - (H.buying_price)) / H.buying_price) * 100) as up_down
FROM holding_stock as H JOIN stock as S ON H.stock_id = S.stock_id
WHERE H.user_id = ? and S.stock_name LIKE ?`
```

앞서 본 쿼리와 거의 유사하지만 마지막에 stock_name을 LIKE로 검사한다.

```
db.query(query2, [req.session.userID, '%' + stockName + '%'], (error, results, fields) => {
  if(error) throw error;
  if(results.length > 0){
    res.render('holdingStockPage', {stocks : results});
  }
  else{
    db.query(query, [req.session.userID], (error, results, fields) => {
      if(error) throw error;
      return res.render('holdingStockPage', {searchValue : 'fail', stocks : results});
    });
  }
});
```

Post로 들어온 stockName 앞 뒤에 '%'를 붙여 부분일치 검색이 가능하도록 구현하였다. 하지만 만약 입력한 stockName과 일치하는 종목명이 없으면 현재 사용자의 모든 보유 주식을 보여주고 실패 메세지를 띄운다.

● UserStockTradingLog (사용자의 주식 거래 내역)

UserInfo 페이지에서 현재 사용자의 모든 주식 거래 내역을 확인하기 위해서는 '주식 거래 내역 보기'를 클릭하면 된다.

```
var query = `SELECT S.stock_name, L.buy_sell, L.transaction_volume, L.transaction_time
FROM stock_trading_log AS L JOIN stock AS S ON L.stock_id = S.stock_id
WHERE L.user_id = ?
`;
```

처음 주식 거래 내역 페이지에 들어간 경우 해당 사용자의 모든 주식 거래 내역을 출력해야 하므로 stock_trading_log relation에서 user_id를 통해 현재 사용자의 주식 거래 내역을 찾는다. 이 과정에서 stock_name 역시 필요하기 때문에 stock과 stock_trading_log를 stock_id를 기준으로 join하였다.

사용자의 주식 거래 내역 역시 종목명으로 부분 일치 검색이 가능하다.

```
var query2 = `SELECT S.stock_name, L.buy_sell, L.transaction_volume, L.transaction_time
FROM stock_trading_log AS L JOIN stock AS S ON L.stock_id = S.stock_id
WHERE L.user_id = ? and S.stock_name LIKE ?
`;
```

Query2는 query와 거의 동일하지만 stock_name을 LIKE를 통해 검색창에 입력된 종목명과 비슷한지 검사한다.

```
router.post('/', (req, res) => {
  const {stockName} = req.body;
  db.query(query2, [req.session.userID, '%' + stockName + '%'], (error, results, fields) => {
    if(error) throw error;
    if(results.length > 0){
      return res.render('userTradingLogPage', {stocks : results});
    }
    else{
      db.query(query, [req.session.userID], (error, results, fields) => {
        if(error) throw error;
        res.render('userTradingLogPage', {searchValue : 'fail', stocks : results});
      });
    }
  })
});
```

stockName 앞 뒤에 '%'를 붙여 부분 일치 검색이 가능하도록 하였다.

● Signout (회원탈퇴)

Menu 페이지에서 회원 탈퇴를 원하면 '회원 탈퇴' 항으로 이동할 수 있다.

```
router.post('/', (req, res) => {
  const {userID, userPassword} = req.body;
  if(userID !== req.session.userID){
    return res.render('signoutPage', {WrongUserInfo : 'wrong'});
  }
  if(userPassword !== req.session.userPassword){
    return res.render('signoutPage', {WrongUserInfo : 'wrong'});
  }
  db.query("DELETE FROM user WHERE user_id = ?", [userID], (error, results, fields) => {
    if(error) throw error;
    return res.render('home', {signoutValue : 'success'});
  });
})
```

회원 탈퇴 항에서는 입력으로 userID와 userPassword를 받는다. 이를 통해 먼저 현재 사용자의 정보와 일치하는지 검사한다. 만약 현재 사용자가 맞다면 user relation에서 현재 사용자를 지우고 home 페이지로 이동한다. 여기서 중요한 점은 milestone 1을 수행할 때 schema 설계 할 때 다른 많은 relation에서 user_id를 foreign key로서 활용하였다. MySQL에서 해당 foreign key에 대해 모두 ON DELETE CASCADE 옵션을 추가해놨다. 이는 user에서 entity 하나가 삭제되면 동일한 user_id를 foreign key로 갖는 다른 모든 relation의 entity들도 삭제된다는 것이다. 즉, 한 user가 삭제되면 그 user의 holding_stock (보유 주식), stock_trading_log (주식 거래 내역) 등이 모두 삭제된다.

● TradingVolumeStatistic (거래량 통계)

```
var query = `SELECT stock_id, stock_name, current_price, trading_volume
FROM stock
ORDER BY trading_volume DESC`
```

거래량의 순위 통계를 내기 위해 stock relation으로 부터 거래량으로 내림차순으로 정렬한 후 데이터를 추출하였다.

● StockInfo (주식 정보)

Menu 페이지에서 모든 주식 종목에 대한 정보를 보고 싶으면 '주식 정보' 버튼을 클린 한다.

```
var query = `SELECT stock_id, stock_name, current_price, closing_price,
    trading_volume, ((current_price - closing_price) / closing_price) * 100 AS price_change_percentage
    FROM stock`
```

처음 주식 정보 페이지에 들어가면 모든 주식에 대한 정보를 가져야 하므로 모든 주식에 대한 정보를 stock relation으로부터 뽑아낸다. 여기서 stock의 attribute들을 이용해 전일 대비 주가 등락률 역시 같이 뽑아낸다.

주식 정보 페이지에서도 종목명 부분일치 검색이 가능하다.

```
router.post('/', (req, res) => {
  const {stockName} = req.body;
  db.query(`SELECT stock_id, stock_name, current_price, closing_price, trading_volume,
    ((current_price - closing_price) / closing_price) * 100 AS price_change_percentage
    FROM stock WHERE stock_name LIKE ?`,
    [`%${stockName}%`], (error, results, fields) => {
      if(error) throw error;
      if(results.length > 0){
        return res.render('stockInfoPage', {stocks : results});
      }
      else{
        db.query(query, (error, total_results, fields) => {
          if(error) throw error;
          res.render('stockInfoPage', {searchValue : 'fail', stocks : total_results});
        })
      }
    }
})
```

종목명이 stockName으로 들어오면 LIKE를 사용한 쿼리를 이용해 해당 종목명과 유사한 종목에 대한 주식 정보를 뽑아낸다. 만약 stockName으로 들어온 종목 이름과 부분일치하는 종목이 없으면 전체 종목에 대한 정보를 출력하면서 해당 종목이 존재하지 않는다는 alert를 보낸다.

● Buy (매수)

주식 정보 페이지에서 특정 종목에 대한 매수를 원하면 '매수' 버튼을 클릭한다. 매수 페이지에서는 수량과 금액을 입력으로 받는다. 금액 옆에 '시장가' 버튼이 있는데 이를 누르면 시장가가 금액란으로 들어가게 구현하였다. 시장가는 다음과 같이 찾았다.

```

router.post('/', (req, res) => {
  const {stockID} = req.body;
  req.session.stockID = stockID;
  var orderPrice = 0;
  db.query(`SELECT order_price
    FROM sell
    WHERE stock_id = ?
    ORDER BY order_price ASC, order_id ASC
    LIMIT 1
  `, [stockID], (error, results, field) => {
    if (error) throw error;
    if(results.length === 0){
      res.render('buyPage', {marketPrice : orderPrice});
      return;
    }
    orderPrice = results[0].order_price;
    res.render('buyPage', {marketPrice : orderPrice});
  });
});

```

매도 주문에서 가장 싸고 가장 오래된 주문에 대해 order_price를 가져온다. 해당 order_price가 시장가이기 때문에 이를 다시 매수 페이지에 넘겨준다. 하지만 만약 매도 주문이 걸려 있는게 없다면 시장가가 없다는 것으로 0이 들어가게 설정하였다.

이제 매수 주문을 볼 차례이다. 이 기능이 가장 핵심적이므로 자세하게 설명하도록 하겠다. 비동기적인 쿼리 실행을 막기 위해 async와 await를 사용하였다.

```

const [marketPrice] = await db.promise().query(`SELECT order_price
  FROM sell
  WHERE stock_id = ?
  ORDER BY order_price ASC, order_id ASC
`, [stockID]);

```

먼저, 매도 주문에서 가장 낮은 가격의 매물을 찾는다.

```

if(marketPrice.length === 0){
  await db.promise().query(`INSERT INTO buy(user_id, stock_id, order_volume, order_type, order_price, order_time)
    VALUES (?,?,?,?,?,?)` , [req.session.userID, req.session.stockID, Amount, 0, price, new Date()]);
}

```

만약 그러한 매물이 없으면 즉시 buy relation, 즉, 매수 주문내역에 매수 주문을 추가해 준다

```
//매수자 예수금 차감
await db.promise().query(`

    UPDATE user
    SET user_deposit = user_deposit - ?
    WHERE user_id = ?

    , [Amount * price, req.session.userID]);
return;
```

매수 주문이 buy relation에 추가되는 순간 매수자의 예수금이 차감된다. (여기서 중요한 것은 매도자의 예수금은 주문이 체결되는 순간 증가하지만 매수자의 예수금은 주문이 체결되는 순간이 아닌 주문이 접수되는 순간 감소한다는 것이다.)

```
//지정가 매수 주문을 걸었는데 그 지정가가 시장가보다 높은 경우는 시장가 거래로 함
if(price >= marketPrice[0].order_price){
    isMarketPrice = 'true';
    price = marketPrice[0].order_price;
}
```

필자는 만약 시장가보다 높은 금액으로 매수주문을 걸었다면 시장가에 거래되도록 구현하였다. 그 이유인 즉슨, 현재가가 너무 크게 변동할 것이기 때문이고 돈이 증발하기 때문이다. 예를 들어, 두 가지 상황을 고려해 볼 수 있다. 매도 시장가가 50000원인 경우 매수 주문을 55000원에 넣었다고 가정해보자. 첫 번째 상황은 55000원에 걸려있는 매도 주문에 거래되는 것이다. 그렇다면 현재가는 갑자기 55000원으로 뛸 것이고 50000원에 걸려있는 매도주문은 체결되지 않은 채 남아있을 것이다. 이는 주식이 아닌 "경매"와 같은 상황이 되는 것이다. 극단적으로 누군가가 현재가 50000원인 주식을 100만원에 매수주문을 넣었다면 그 주식을 100만원에 매도 주문을 건 사람은 마치 로또에 당첨된 것과 같은 상황인 것이다. 또한 현재가가 50000원에서 갑자기 100만원이 될 것이다. 이 상황이 필자는 불가능 하다고 생각하여 시장가보다 높은 금액으로 매수 주문을 걸면 시장 가에 거래되도록 구현하였다. 두 번째 상황은 50000원에 매도 시장가가 걸려있는 상태에서 55000원에 매수 주문을 걸 때 50000원에 결제 되는 경우이다. 이 경우 역시 매수자가 5000원일 손해보는 일이 일어난다. 이 5000원은 행방을 알 수가 없다. 극단적으로 예를 들어 보면, 50000원에 매도 시장가가 걸려 있는 주식에 대해 100만원 매수주문을 걸면 95만원의 행방은 알 수 없게 된다. 이 역시 불가능한 상황이라 생각하여 필자는 시장

가보다 매수 주문의 가격이 더 클 경우 시장가에 거래되도록 설정하였다.

시장가 주문인 경우 입력한 수량이 바닥 날 때 까지 다음을 수행한다.

```
//먼저 시장가 tuple 꺼내기 여기서 중요한 것은 시장가 주문이기에 그냥 가장 적고 오래된 금액을 찾으면 됨
var [sellOrder] = await db.promise().query(`

    SELECT *
    FROM sell
    WHERE stock_id = ? and order_price = ?
    ORDER BY order_id ASC
    LIMIT 1
`, [stockID, price]);
```

먼저 시장가와 대응되는 tuple을 매도 주문에서 꺼낸다.

```
//남아있는 시장가 최소 금액 매도 주문이 없을 때
if(sellOrder.length === 0){
    //최소 금액 매도 주문 찾기
    [sellOrder] = await db.promise().query(`

        SELECT *
        FROM sell
        WHERE stock_id = ?
        ORDER BY order_price ASC, order_id ASC
        LIMIT 1
    `, [stockID]);
```

만약 시장가에 걸려있는 주문이 없다면 시장가 다음으로 낮은 금액의 매도 주문을 찾는다.

```
//진짜 남아있는 매도 주문이 없을 때
if(sellOrder.length === 0){
    //매수 주문 내역에 추가하고 끝
    await db.promise().query(`

        INSERT INTO buy(user_id, stock_id, order_volume, order_type, order_price, order_time)
        VALUES (?,?,?,?,?,?)

    `, [req.session.userID, stockID, Amount, 0, price, new Date()])
```

그 마저도 존재하지 않는다면 buy relation에 매수 주문을 추가해주고 매수자의 예수금을 깎는다.

이렇게 구현을 한 이유는 다음과 같은 상황 때문이다. 만약 매도 시장가에 10주가 걸려 있고 매수 주문을 시장가에 15주 걸었다고 하자. 그러면 시장가에 10주가 팔리고 나머지 5주는 시장가보다는 조금 더 높은 가격에 매수가 될 것이다. 이렇게 순차적으로 매도 주

문을 가겨울 타고 올라가면서 매수 주문이 체결될 것이다. 때문에 다음과 같이 구현하였다.

시장가에 매도 주문이 걸려 있는 상황에서는 세 가지 상황을 고려하여 구현하였다.

먼저, 매수하려는 수량이 매도 주문에 걸려 있는 수량보다 많은 경우이다.

```
Amount -= order_volume;

//현재 tuple을 매도 주문 relation에서 제거하고 주식 거래 내역 relation으로 보내기
await db.promise().query(`DELETE FROM sell
    WHERE order_id = ?
    `, [order_id]);

await db.promise().query(`INSERT INTO stock_trading_log(user_id, stock_id, buy_sell, transaction_volume, transaction_time)
    VALUES (?, ?, ?, ?, ?)
    `, [user_id, stock_id, order_price, order_volume, new Date()]);

//매도자 예수금 증가시키기
await db.promise().query(`UPDATE user
    SET user_deposit = user_deposit + ?
    WHERE user_id = ?
    `, [order_volume * order_price, user_id]);

//매도자의 보유주식 업데이트
await update_seller_holding_stock(user_id, stock_id, order_volume);
```

이 경우 매수하려는 수량을 매도 주문에 걸려 있는 수량만큼 감소시키고 매도 주문을 체결해준다. 먼저 매도 주문에서 해당 주문을 지우고 주식 거래 내역에 매도 주문 내역을 추가해준다. 매수 주문이 체결되었으므로 매도자의 예수금을 증가시켜준다. 또한 매도자의 보유 주식이 감소하였을 것이므로 매도자의 holding_stock를 업데이트 시켜준다. 이 함수에 대해서는 추후 자세히 살펴보겠다.

```
//매수자 거래내역 추가
await db.promise().query(`INSERT INTO stock_trading_log(user_id, stock_id, buy_sell, transaction_volume, transaction_time)
    VALUES (?, ?, ?, ?, ?)
    `, [req.session.userID, req.session.stockID, -order_price, order_volume, new Date()]);

//매수자 예수금 감소시키기
await db.promise().query(`UPDATE user
    SET user_deposit = user_deposit - ?
    WHERE user_id = ?
    `, [order_volume * order_price, req.session.userID]);

//매수자 보유주식 업데이트
await update_buyer_holding_stock(req.session.userID, req.session.stockID, price, order_volume);
//현재가 업데이트
await db.promise().query(`UPDATE stock
    SET current_price = ?
    WHERE stock_id = ?
    `, [price, stock_id]);
```

매수자의 경우 매수 주문이 일부 체결된 것이므로 체결된 매수 주문에 한하여 주식 거래 내역에 체결된 매수 주문을 넣어준다.

여기서는 매수자의 예수금을 감소시켜 주는데, 이는 구현시 매도와 매수를 두 파일로 구분하여 구현하였기 때문이다. 매수의 경우 지금 상황은 매수가 즉시 체결되는 상태이기 때문에 매수 주문이 buy relation에 들어간 적이 없다. 즉, 매수자의 예수금이 깎인 적이 없는 것이다. (즉시 체결되는 상황이므로) 때문에 체결과 동시에 예수금을 깎아준다. 또한 매수자의 보유 주식이 증가하였을 것이므로 이 역시 업데이트 해준다. 마지막으로, 거래가 체결된 순간 현재가도 바뀌기에 현재가 역시 새로운 값으로 수정해준다.

```
//거래가 체결된 것으로 거래량 업데이트
await db.promise().query(`  
    UPDATE stock  
        SET trading_volume = trading_volume + ?  
        WHERE stock_id = ?  
    `, [order_volume, stock_id]);
```

거래가 체결된 상황이므로 거래된 수량만큼 해당 종목의 거래량을 늘려준다.

여기서 좀 정리를 하고 가자면, 필자가 매수와 매도를 구현할 때 이와 같은 생각을 했다. 어차피 모든 거래는 시장가에 될 것이다. 이 말인 즉슨, 모든 거래는 "시장가"에 체결된다는 것이다. 매수 주문이 지정가에 들어가더라도 이가 체결되는 시점은 그 지정가가 매수 시장가가 됐을 때이다. 매도 역시 마찬가지다. 매도 지정가 주문을 넣더라도 매도 주문이 체결되는 순간은 그 지정가가 시장가가 됐을 때이다. 때문에 지금 매수 구현에서는 매도 시장가보다 낮은 금액은 그냥 buy relation에 추가해 주고 웬만한 경우는 즉시 체결을 시킨다는 것이다.

```

else if (Amount < order_volume){
    //매도 주문에서 수량을 제거
    await db.promise().query(` 
        UPDATE sell
        SET order_volume = order_volume - ?
        WHERE order_id = ?
    `, [Amount, order_id]);

    //매도 주문 체결된 것 주식 거래 내역에 추가
    await db.promise().query(` 
        INSERT INTO stock_trading_log(user_id, stock_id, buy_sell, transaction_volume, transaction_time)
        VALUES (?, ?, ?, ?, ?)
    `, [user_id, stock_id, order_price, Amount, new Date()]);
}

//매도자 주문 체결된 주식 예수금에 증가시키기
await db.promise().query(` 
    UPDATE user
    SET user_deposit = user_deposit + ?
    WHERE user_id = ?
`, [Amount * order_price, user_id]);

//매도자 보유주식 업데이트
await update_seller_holding_stock(user_id, stock_id, Amount);

```

다음으로는 매도 잔량이 매수 수량보다 많은 경우이다. 이 경우 모든 매수 주문이 체결될 것이고 매도 주문의 일부만 체결될 것이다. 때문에 먼저 매도 주문의 매도 수량을 매수 수량만큼 뺀 값으로 업데이트 해준다. 체결된 거래에 대해 stock_trading_log에 해당 거래 내역을 넣어준다. 매도자의 보유주식과 예수금의 변동이 생겼으므로 이 역시 업데이트 해준다.

```

//매수자 거래내역 추가
await db.promise().query(` 
    INSERT INTO stock_trading_log(user_id, stock_id, buy_sell, transaction_volume, transaction_time)
    VALUES (?, ?, ?, ?, ?)
`, [req.session.userID, req.session.stockID, -order_price, order_volume, new Date()]);

//매수자 예수금 감소시키기
await db.promise().query(` 
    UPDATE user
    SET user_deposit = user_deposit - ?
    WHERE user_id = ?
`, [order_volume * order_price, req.session.userID]);

//매수자 보유주식 업데이트
await update_buyer_holding_stock(req.session.userID, req.session.stockID, price, order_volume);
//현재가 업데이트
await db.promise().query(` 
    UPDATE stock
    SET current_price = ?
    WHERE stock_id = ?
`, [price, stock_id]);

//거래가 체결된 것으로 거래량 업데이트
await db.promise().query(` ...
    UPDATE stock
    SET trading_volume = trading_volume + ?
    WHERE stock_id = ?
`, [order_volume, stock_id]);

```

Amount > order_volume인 상황이랑 동일하게 매수자의 정보가 업데이트 된다. 하지만 하나 다른 것은 여기서는 Amount -= order_volume 되는 것이 아닌 order_volume -= Amount가 되는 것이다.

마지막 상황은 매도 잔량과 매수 수량이 동일한 경우이다. 이 역시 동일한 매커니즘으로 수행되기에 설명은 생략하겠다.

정리하면 다음과 같다.

매수 주문 발생 => 매도 시장가보다 크면 시장가 주문으로 취급 / 아니면 지정가 주문 => 시장가 주문인 경우 시장가에 매도 잔량이 얼마나 남았는 지에 따라 계속 최소 매도 가를 찾으며 거래 수행. => 만약 더이상 남은 최소 매도가의 매도 잔량이 없다면 buy relation에 남은 매수 주문 수량만큼을 추가 (이 주문의 체결은 매도 주문이 걸렸을 때 하게 됨)

=> 지정가의 경우 buy relation에 추가 (이 주문의 체결은 매도 주문이 걸렸을 때 하게 됨)

● Sell (매도)

매도의 경우 매수와 정반대로 이루어 진다. 때문에 코드를 보며 설명하지 않고 매커니즘을 설명하도록 하겠다.

매도 주문 발생 => 매수 시장가보다 작으면 시장가 주문으로 취급 / 아니면 지정가 주문 => 시장가 주문인 경우 시장가에 매수 잔량이 얼마나 남았는 지에 따라 계속 최대 매수가를 찾으며 거래 수행. => 만약 더이상 남은 최대 매수가의 매수 잔량이 없다면 sell relation에 남은 매도 주문 수량만큼을 추가 (이 주문의 체결은 매수 주문이 걸렸을 때 하게 됨)

=> 지정가의 경우 sell relation에 즉시 추가 (이 주문의 체결은 매도 주문이 걸렸을 때 하게 됨)

Buy와 sell 모두에서 활용된 update_seller_holding_stock 함수와 update_buyer_holding stock 함수에 대해 살펴보자.

이것이 호출되는 시점은 주문이 체결될 때이다.

```
// 매도자 보유 주식 감소시키기
const update_seller_holding_stock = async (user_id, stock_id, order_volume) => {
  try {
    const [results] = await db.promise().query(`SELECT holding_num
      FROM holding_stock
      WHERE user_id = ? AND stock_id = ?
    `, [user_id, stock_id]);

    if(results.length === 0){
      throw error;
    }

    const holding_num = results[0].holding_num;

    if (holding_num > order_volume) {
      // 보유 주식이 거래량보다 많을 때 보유 주식 수와 거래 가능 주식 수 감소시키기
      await db.promise().query(`UPDATE holding_stock
        SET holding_num = holding_num - ?
        WHERE user_id = ? AND stock_id = ?
      `, [order_volume, user_id, stock_id]);
    } else if (holding_num === order_volume) {
      // 보유 주식이 거래량과 같을 때 보유 주식 삭제
      await db.promise().query(`DELETE FROM holding_stock
        WHERE user_id = ? AND stock_id = ?
      `, [user_id, stock_id]);
    }
  } catch (error) {
    throw error;
  }
};
```

먼저 매도자의 보유 주식을 업데이트 하는 함수이다. 먼저 매도자의 거래하는 주식에 대한 보유 주식을 tuple로 가져온다. 보유 주식 수가 주문 체결된 수량보다 많으면 보유 주식을 tuple로서 계속 남겨놔야 하므로 holding_stock를 update시켜준다. 하지만 만약 보유 주식 수가 주문 체결된 수량과 같으면 해당 주식을 더이상 보유하지 않게 되는 것으로 해당 보유 주식 tuple을 삭제한다.

```

const update_buyer_holding_stock = async (user_id, stock_id, price, order_volume) => {
  try{
    const [results] = await db.promise().query(` 
      SELECT holding_num, buying_price
      FROM holding_stock
      WHERE user_id = ? and stock_id = ?
    `, [user_id, stock_id]);

    //해당 주식 보유수량이 0개일때
    if(results.length === 0){
      //새로운 tuple 추가해줌
      await db.promise().query(` 
        INSERT INTO holding_stock(user_id, stock_id, buying_price, transaction_available, holding_num)
        VALUES (?, ?, ?, ?, ?)
      `, [user_id, stock_id, price, order_volume, order_volume]);
    }
    //해당 주식을 보유하고 있을 때
    else{
      //값을 갱신해줌. 중요한 것은 buying_price를 비율에 맞게 해준다는 것
      const new_buying_price = parseFloat(
        (
          (parseInt(results[0].buying_price) * parseInt(results[0].holding_num) + price * order_volume) /
          (parseInt(results[0].holding_num) + parseInt(order_volume))
        ).toFixed(2)
      );

      await db.promise().query(` 
        UPDATE holding_stock
        SET holding_num = holding_num + ?, transaction_available = transaction_available + ?, buying_price = ?
        WHERE user_id = ? and stock_id = ?
      `, [order_volume, order_volume, new_buying_price, user_id, stock_id]);
    }
  }catch (error){
    throw error;
  }
}

```

다음으로 매도자의 보유 주식 내역을 업데이트 하는 함수이다. 매수한 주식을 원래 가지고 있지 않으면 holding_stock에 새로운 tuple을 추가시켜준다. (새로운 종목을 보유하게 된 것이므로) 만약 매수한 주식을 이미 몇 주 보유한 상태라면 보유 주식을 update 시켜 준다. 여기서 보유 주식의 매수가는 원래 있던 주식의 매수가와 매수한 주식의 매수가를 비율에 맞춰 평균내어 구하였다.

● StockTradingLog (주식 거래 내역)

주식 종목에 대한 거래 내역을 확인하기 위해서는 주식 정보에서 종목에 대해 '주식 거래 내역'을 클릭하면 된다.

```
var query =  
SELECT  
    buyer.user_id AS buyer_id,  
    seller.user_id AS seller_id,  
    ABS(seller.buy_sell) AS price,  
    buyer.transaction_volume AS transaction_volume,  
    buyer.transaction_time AS transaction_time  
FROM  
    stock_trading_log AS buyer  
JOIN  
    stock_trading_log AS seller ON  
    buyer.stock_id = seller.stock_id AND  
    buyer.transaction_volume = seller.transaction_volume AND  
    buyer.transaction_time = seller.transaction_time AND  
    buyer.buy_sell < 0 AND  
    seller.buy_sell > 0  
JOIN  
    stock AS S ON buyer.stock_id = S.stock_id  
WHERE  
    S.stock_id = ?  
ORDER BY  
    buyer.transaction_time  
`
```

매수자의 ID와 매도자의 ID 그리고 거래량 등을 모두 뽑아내기 위해 stock_trading log과 stock을 join하였다. 또한, 앞서 매수에서 확인한 바와 같이 한 번 거래가 체결되면 두 개의 기록이 남는다. (하나는 매수기록, 하나는 매도 기록) 때문에 두 기록을 하나로 합치는 작업을 수행하였다. (stock_trading_log와 stock_trading_log를 join) stock_id를 통해 선택한 종목에 대한 거래 내역만을 추출한다.

● OrderBook (호가창)

호가창의 경우 먼저 해당 종목에 대한 매도 주문과 매수 주문을 모두 가져온다.

```
const [sellOrder] = await db.promise().query(`  
    SELECT order_price, sum(order_volume) as sell_volume  
    FROM sell  
    WHERE stock_id = ?  
    GROUP BY order_price  
    ORDER BY order_price DESC  
`, [stockID]);  
  
const [buyOrder] = await db.promise().query(`  
    SELECT order_price, sum(order_volume) as buy_volume  
    FROM buy  
    WHERE stock_id = ?  
    GROUP BY order_price  
    ORDER BY order_price DESC  
`, [stockID]);
```

그 후 stock의 price_unit을 활용하여 호가창 내에 보이는 주문들을 선별한다.

```
for(var i = 0; i < 10; i++){  
    const Price = current_price + (5-i) * price_unit;  
  
    const sell_order = sellOrder.find(order => order.order_price === Price);  
    sellOrders.push({price : Price, volume : (sell_order ? sell_order.sell_volume : '-')});  
  
    const buy_order = buyOrder.find(order => order.order_price === Price);  
    buyOrders.push({price : Price, volume : (buy_order ? buy_order.buy_volume : '-')});  
}  
res.render('orderBookPage', {stock_name, current_price, price_unit, sellOrders, buyOrders});
```

(5단계 호가창이므로 현재가로부터 위아래로 5단계)

Trouble Shooting

본 과제를 수행하면서 다양한 문제를 겪었다. 대부분은 구현에서의 디테일한 고민이었지만 한 가지 정말 큰 문제가 있었다.

매수자의 holding_stock을 변화시키는 과정에서 문제가 있었다. 예를 들어 60000원에 10주를 먼저 매수한 다음 65000에 10주를 매수하면 분명 매수가는 62500이 되어야 할 것이다.(비율에 맞게 평균을 내게 수식을 설정하였기 때문이다.) 하지만 매수가가 1238원이 나와 이를 디버깅하기 위해 정말 많이 애썼다.

```
//값을 정한바로 중요한 것은 buying_price를 비율에 맞게 바꾼다는 것
const newBuyingPrice = parseFloat(
  (
    (parseInt(results[0].buying_price) * parseInt(results[0].holding_num) + price * order_volume) /
    (parseInt(results[0].holding_num) + parseInt(order_volume))
  ).toFixed(2)
);
```

이 수식이 바로 그것이다. 첨부된 사진은 오류를 수정한 코드이기에 문제가 없지만 원래는 parseInt와 parseFloat가 존재하지 않았다.

1238원이 나온 이유는 다음과 같았다. 분모의 result[0].holding_num + order_volume에서 result[0].holding_num이 object로부터 나왔기에 string type이었던 것이다. 때문에 '10' + 10 = 1010 이 되어 1238원이 나오게 되었다.

이 문제를 해결하기 위해 모든 값을 console.log로 확인해보면서 디버깅을 수행하였다.

추가적으로 하고 싶은 말:

이 과제를 수행하면서 정말 많은 노력과 시간을 쏟았습니다. 이를 위키에 모두 담기에는 너무나 많은 분량이 필요하기에 좀 간략하게 적은 감이 없지 않아 있습니다. 영상에서 자세히 설명할 것이기에 영상을 보고 좀 더 자세하게 이해해 주셨으면 합니다. 감사합니다.

시연 영상 링크:

<https://youtu.be/WIzGjuZKDeU>