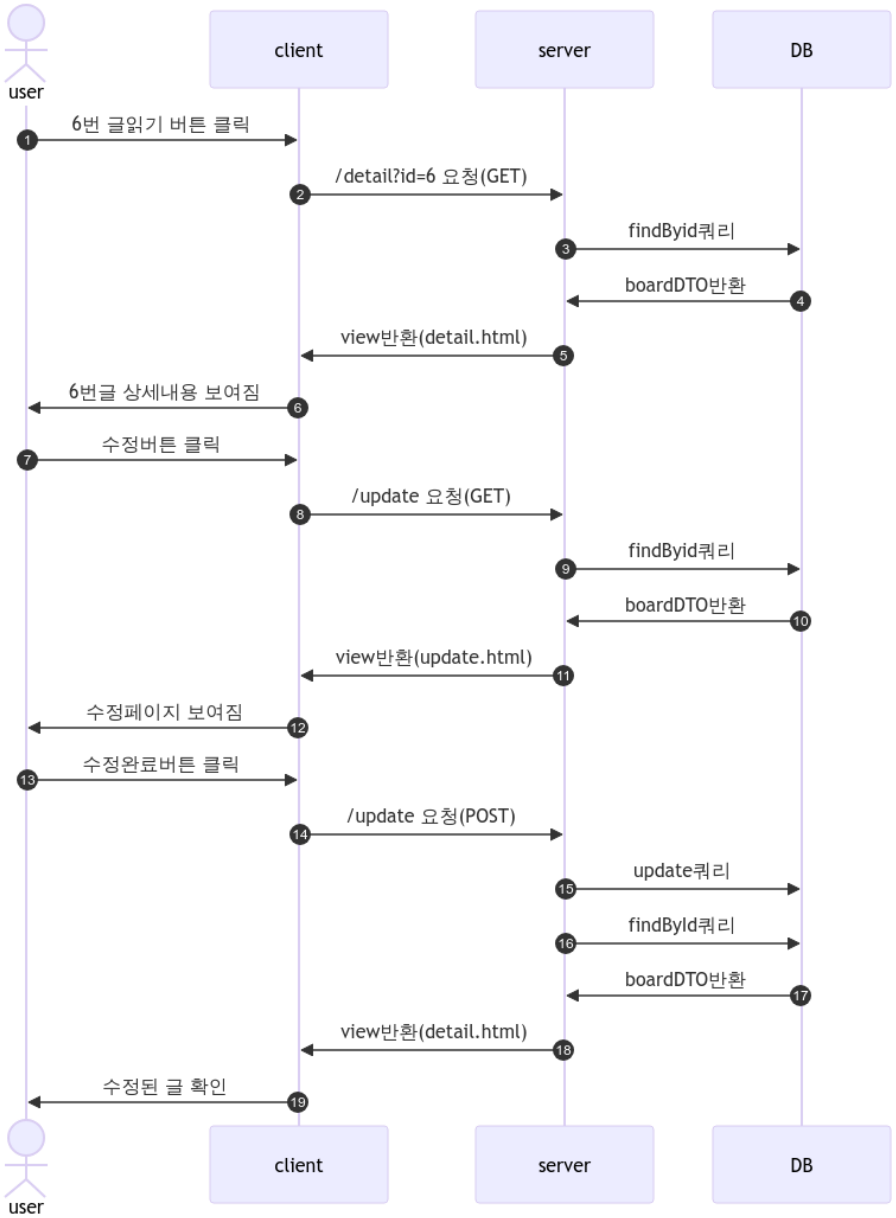
수정

수정은 사실 DB에 이미 작성된 내용을 수정하는 것이라 간단한것 같지만 화면과 ux까지 고려한다면 CRUD 중 가장 복잡한 프로세스를 갖는다. 말로 설명하면 매우 장황해지기 때문에 다이어그램을 통해 살펴보자.

1) 수정 프로세스

위 그림은 사용자가 글 목록에서 상세 글을눌러서 상세조회하는 것부터 시작해서 수정을 완료한 이후 수정된 글 내용을 확인하는 시점까지의 프로세스를 담은 다이어그램이다. 잘 살펴보면 수정 작업을 위해 **상세조회가 총 3번** 일어나는 것을 확인할 수 있다.



update 프로세스

***user: 사용자**

***client: 브라우저, 화면**

***server: springboot 서버**

***DB: mysql DB**

순서에 따라서 하나씩 살펴보자.

1. 사용자가 글목록에서 id가 6인 글을 클릭한다.

2. 상세조회를 위해 브라우저는 서버에게 localhost:8080/6 요청을 보낸다.

3. 서버는 detail로 들어온 요청을 받아 BoardController의 **findById(상세조회)**라는 메서드로 넘겨주고 Controller -> Service -> Repository -> DB 순서로 요청을 처리한다.

4. DB는 해당하는 데이터를 찾아서 BoardDTO에 담아 반환한다.

5. 컨트롤러는 적절한 뷰(detail.html)를 찾아서 반환한다.

6. 사용자는 6번 글의 상세 내용을 확인한다.

7. 사용자가 글을 수정하기 위해 수정버튼을 클릭한다.

8. 서버로 /update/6(POST) 요청을 보낸다.

9. 수정을 할 수 있는 페이지에서 글의 원래 내용을 다시 보여줘야 하기 때문에 **findById(상세조회) 메서드를** 이용해 글의 상세내용을 한 번 더 요청한다.

10. 4번과 동일

11. detail.html 뷰를 반환한다.

12. 사용자는 수정페이지를 확인한다.

13. 사용자는 수정할 내용을 기입하고 완료버튼을 클릭한다. 이때 글 쓴 사용자가 맞는지 여부를 확인하는 로직이 들어간다.

***이번 프로젝트에서는 글쓴이를 식별하기 위해 글을 작성할때 기입한 비밀번호(boardPass)와 수정할 때의 비밀번호가 일치하는지를 확인하였다. 매우 조악한 방법이지만 다른 방식으로 구현하려면 사용자 인증여부를 확인하는 기능이 필요하다. 본 프로젝트는 게시판의 최소기능만을 구현하기 때문에 이 방식으로 진행하였다.**

14. 서버로 /update/6(POST)요청을 보낸다.

15. update메서드를 실행한다.(DB에 수정사항 반영)

16. DB에 수정된 내용이 반영되었으니 다시 상세글 내용을 불러와야 한다. 때문에 **findById(상세조회) 메서드를** 한번 더 실행한다.

17. 4번과 동일

18. 수정된 글내용을 보여줘야 하기 때문에 컨트롤러는 detail.html 뷰를 반환한다.

19. 사용자는 최종적으로 수정완료된 글을 확인한다.

2) 수정기능 구현

#BoardController

```
    //수정버튼 클릭시 수정화면으로 넘어가도록 하는 메서드(GET)
    @GetMapping("/update/{id}")
     public String update(@PathVariable("id") Long id, Model model){
          BoardDTO boardDTO = boardService.findById(id);
          model.addAttribute("board", boardDTO);
          return "update";
     }
     //DB에 실절적으로 수정내용을 요청하는 메서드(POST)
     @PostMapping("/update/{id}")
     public String update(BoardDTO boardDTO, Model model){
```

```
        //update요청
        boardService.update(boardDTO);
        //findById로 수정된 내용을 다시조회
        BoardDTO dto = boardService.findById(boardDTO.getId());
        model.addAttribute("board", dto);
        return "detail";
    }
```

수정화면을 보여줄 update(GET) 메서드와 update(POST) 메서드를 구현했다. 글작성 메서드인 save와 마찬가지로 오버로딩을 이용해 같은 메서드이름으로 다른 기능을 구현하였다.

GET방식의 update요청은 상세조회를 한 뒤 수정화면에 해당 글 내용을 띄워준다.

POST방식의 update요청은 실질적으로 글 내용을 수정하는 작업을 DB에 요청하고 수정된 내용을 다시 조회하여 상세조회화면을 사용자에게 보여준다.

#BoardService

```
    public void update(BoardDTO boardDTO) {
        boardRepository.update(boardDTO);
    }
```

#BoardRepository

```
    public void update(BoardDTO boardDTO) {
        sql.update("Board.update", boardDTO);
    }
```

#board-mapper

```
    <update id="update" parameterType="board">
        update board_table
        set boardTitle=#{boardTitle}, boardContents=#{boardContents}
        where id=#{id}
    </update>
```

#detail.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>detail</title>
    <style>
        table, tr, td, th {
            border: 1px solid black;
            border-collapse: collapse;
        }
        th, td {
            padding: 10px;
        }
    </style>
</head>
```

```
<body>
<table>
    <tr>
        <th>id</th>
        <td th:text="${board.id}"></td>
    </tr>
    <tr>
        <th>title</th>
        <td th:text="${board.boardTitle}"></td>
    </tr>
    <tr>
        <th>writer</th>
        <td th:text="${board.boardWriter}"></td>
    </tr>
    <tr>
        <th>date</th>
        <td th:text="${board.createdAt}"></td>
    </tr>
    <tr>
        <th>hits</th>
        <td th:text="${board.boardHits}"></td>
    </tr>
    <tr>
```

```
            <th>contents</th>
            <td th:text="${board.boardContents}"></td>
        </tr>
    </table>
    <button onclick="listReq()">목록</button>
    <button onclick="updateReq()">수정</button>
    <button onclick="deleteReq()">삭제</button>
    </body>
    <script th:inline="javascript">
        const listReq = () => {
            location.href = "/list";
        }
        //수정버튼 클릭시 실행되는 함수
        const updateReq = () => {
            location.href = `/update/[[${board.id}]]`;
        }
        //삭제버튼 클릭시 실행되는 함수
        const deleteReq = () => {
            location.href = `/delete/[[${board.id}]]`;
        }
    </script>
    </html>
```

지난 시간에 작성한 detail.html에서는 수정/삭제 버튼을 클릭했을 때 작동할 함수의 구현부가 비어있었기 때문에 location.href로 서버에 요청을 보내도록 내용을 채워준다.

#update.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<form th:action="@{|/update/${board.id}|}" method="post" name="updateForm">
    <input type="hidden" name="id" th:value="${board.id}">
    writer: <input type="text" name="boardWriter" th:value="${board.boardWriter}" readonly><br>
    title: <input type="text" name="boardTitle" th:value="${board.boardTitle}"><br>
    pass: <input type="text" name="boardPass" id="board-pass"><br>
    contents: <textarea name="boardContents" cols="30" rows="10" th:text="${board.boardContents}"></textarea><br>
    <input type="button" value="수정" onclick="board_update()">
</form>
</body>
<script th:inline="javascript">
    const board_update = () => {
        const boardPass = document.getElementById("board-pass").value;
        const passDB = [[${board.boardPass}]];
```

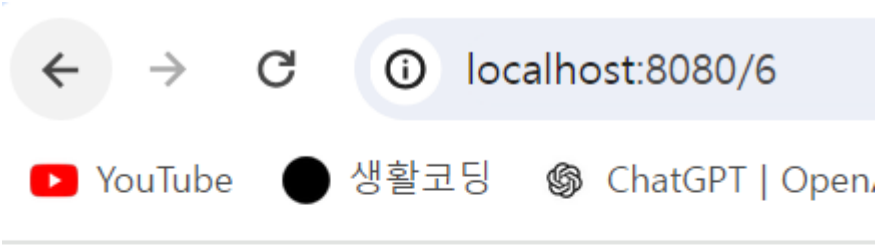
```
        if (boardPass == passDB) {
            updateForm.submit();
        } else {
            alert("비밀번호가 틀립니다!!");
        }
    }
</script>
</html>
```

기존에 만들어뒀던 html파일들과 같은 경로에 update.html을 추가해 준다.

form태그를 이용해 post요청을 보낸다. 버튼을 클릭했을 때 작동하는 board_update함수는 post요청을 submit 하기 전에 현재 글의 비밀번호와 사용자가 입력한 비밀번호가 일치하는지 검증하고 둘의 값이 일치하는 경우는 서버로 요청을 보내고 일치하지 않는 경우 알림 창을 띄워 "비밀번호가 틀립니다!!"라는 메시지를 내보낸다. 디자인적인 요소는 최소한으로 하였기 때문에 흐름만 확인하도록 한다.

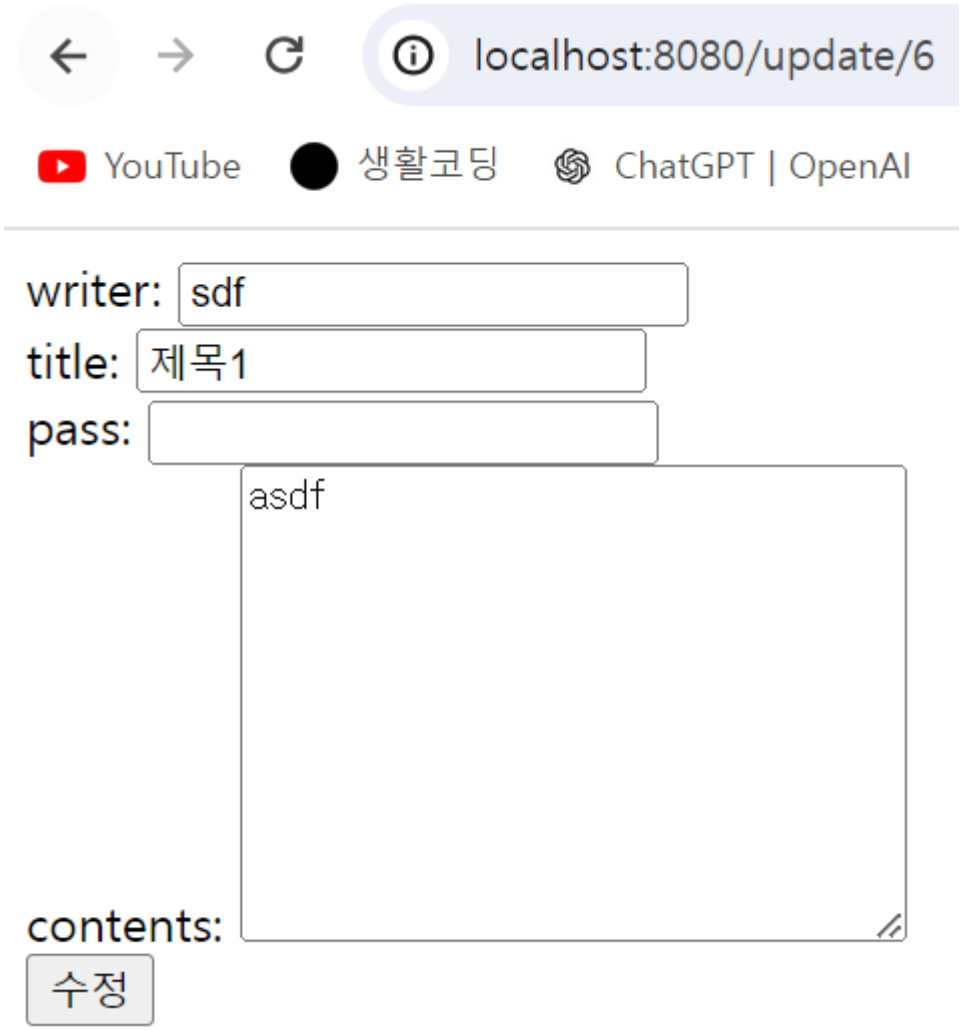
3) 수정기능 테스트

화면에서 테스트를 해보자.



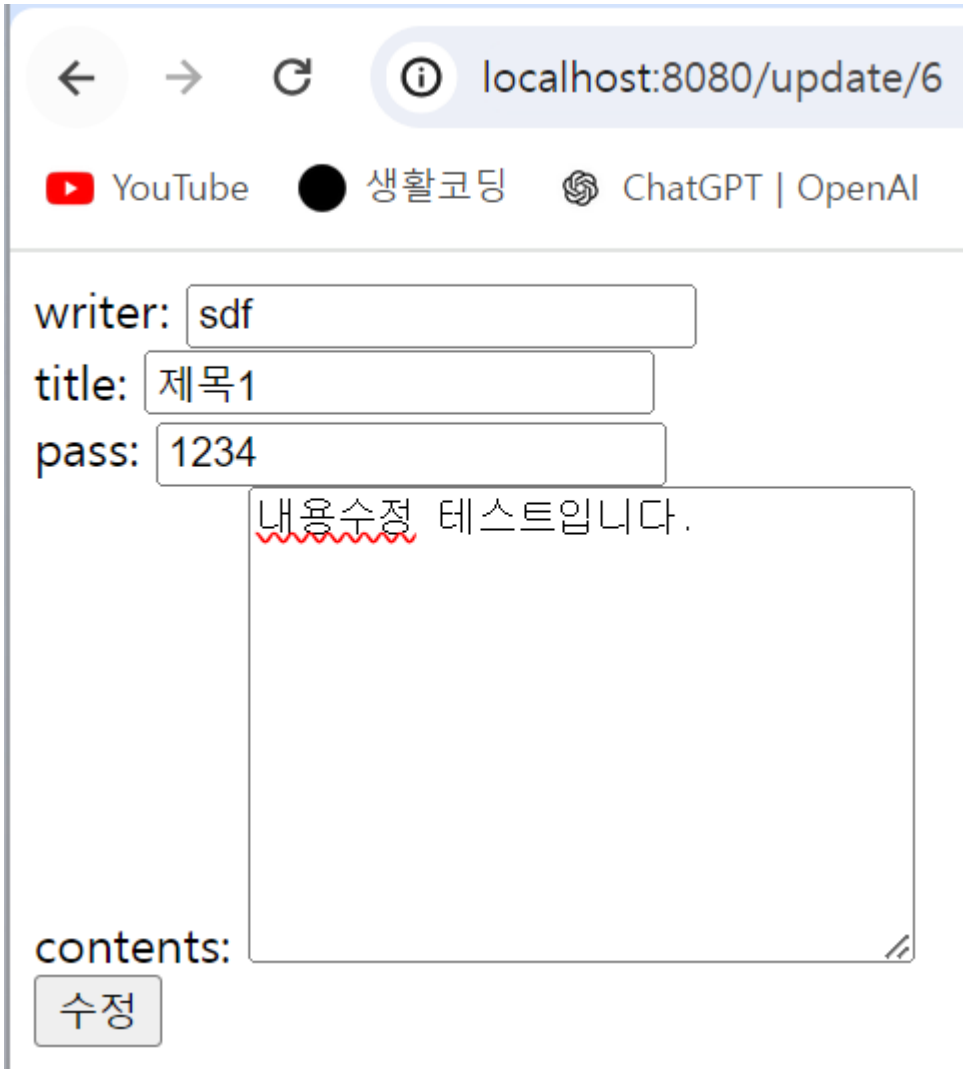
id	6
title	제목1
writer	sdf
date	2024-06-11
hits	3
contents	asdf

목록 수정 삭제



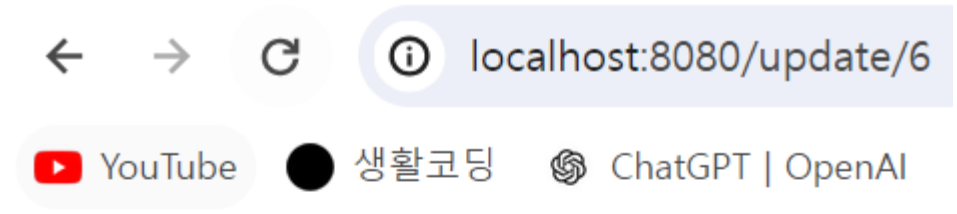
상세조회,수정화면 이동

수정 버튼을 누르니 수정화면으로 페이지가 이동한다.


localhost:8080/update/6
YouTube
생활코딩
ChatGPT | OpenAI
writer: sdf
title: 제목1
pass: 1234
내용수정 테스트입니다.
contents:
수정

수정요청 후 수정반영된 상세조회

내용을 변경하고 "수정" 버튼을 누르면 서버로 수정요청이 넘어가고 서버 쪽에서 처리한 뒤에 수정된 상세화면을 반환한다.



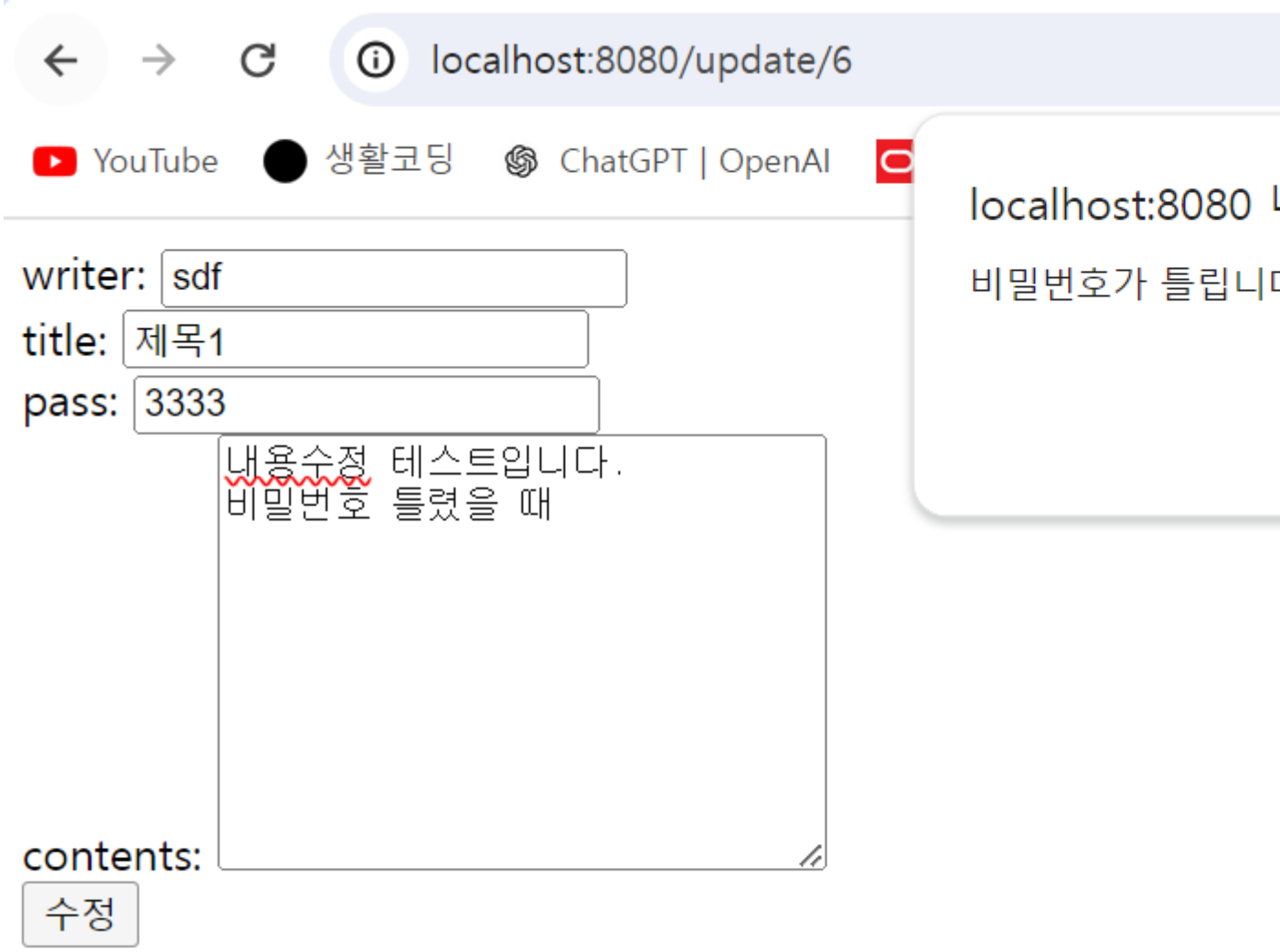
writer: sdf

title: 제목1

pass: 3333

contents: 내용수정 테스트입니다.
비밀번호 틀렸을 때

수정



비밀번호가 틀리면 알림창 호출

비밀번호가 작성 때 기입한 것과 다를 때는 알림 창이 호출된다.

삭제

삭제는 지금까지 기능 중 가장 간단하다. 요청을 넘기고 DB에 반영하여 데이터를 삭제한 뒤 다시 목록화면만 보여주면 끝이다. 물론 실제 서비스에서는 DB에서 데이터를 바로 삭제하지 않고 상태값만 바꾸어 보존한 뒤 사용자에게 노출되지 않도록 구현하기 때문에 더 복잡할 수 있다.

1) 삭제기능 구현

#BoardController

```
@GetMapping("/delete/{id}")

public String delete(@PathVariable("id") Long id){
```

```
        boardService.delete(id);

        return "redirect:/list";

    }
```

요청을 받아 DB로 요청을 넘겨주면 된다. 이때 return값을 "redirect:/list"로 하는 이유는 "/list"로 요청하게 되면 정적인 페이지를 반환하여 글목록이 보이지 않기 때문이다. delete메서드는 db로 삭제 sql만 요청한다. 그다음 다시 글 목록을 받아오는 로직은 포함되어 있지 않다. 때문에 다시 글 목록을 불러오기 위해서는 요청이 새로 이루어져야 하고 이를 위해 redirect를 사용한다.

#BoardService

```
    public void delete(Long id) {

        boardRepository.delete(id);

    }
```

#BoardRepository

```
    public void delete(Long id) {

        sql.delete("Board.delete", id);

    }
```

#board-mapper

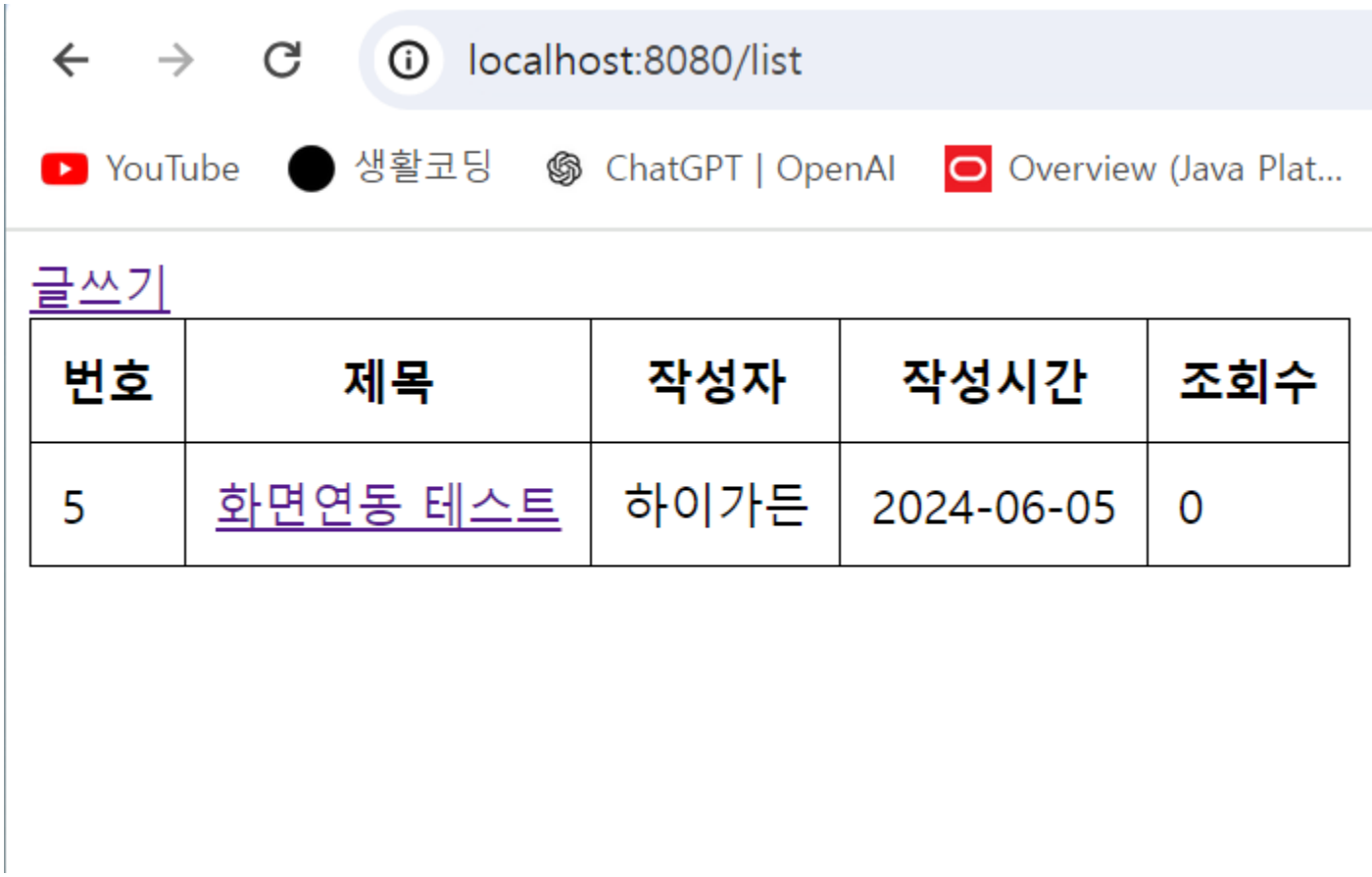
```
    <delete id="delete" parameterType="Long">

        delete from board_table

        where id=#{id}

    </delete>
```

컨트롤러 이후의 로직은 특별한 것이 없다. id에 해당하는 data를 삭제요청하는 것이 끝이다. 따로

화면을 만들 필요도 없다.

2) 삭제기능 테스트

마지막으로 삭제기능을 테스트해보자.



삭제기능 테스트

상세화면에서 삭제버튼을 누르자 삭제된 후 다시 목록이 조회되는 모습을 확인할 수 있다.

기본적인 CRUD기능이 마무리되었다. 다음시간에는 insert기능을 조금 보완하여 파일첨부기능을 추가해보도록 하겠다.