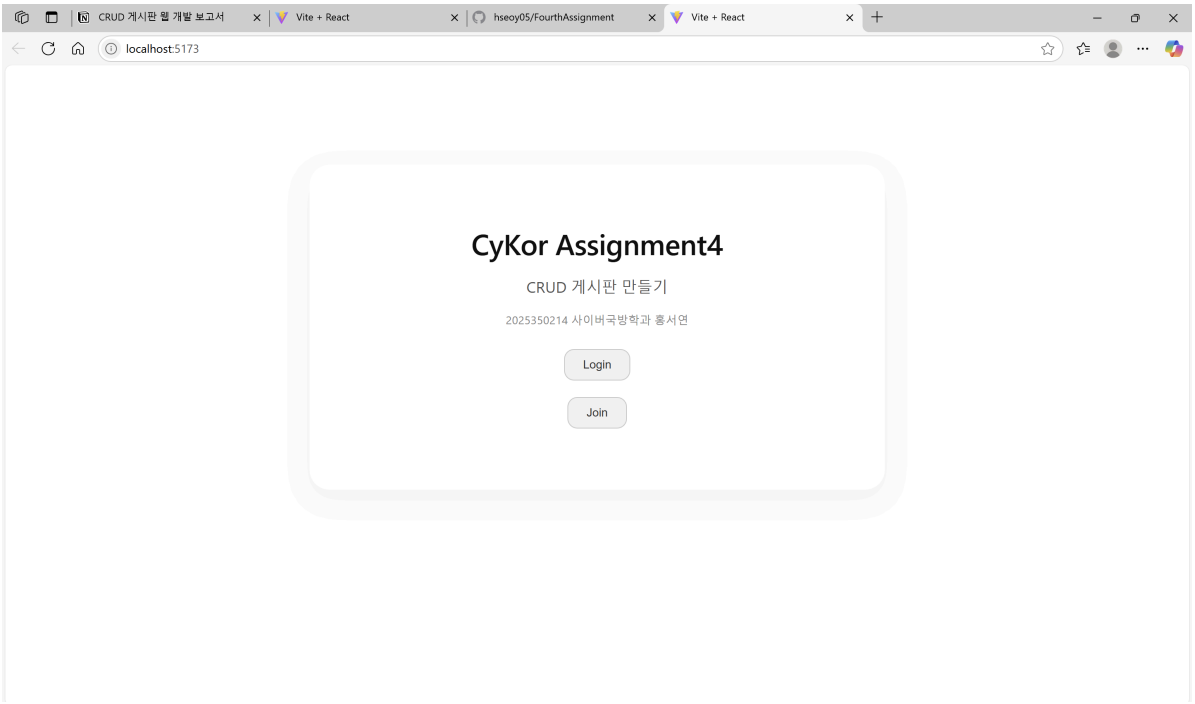
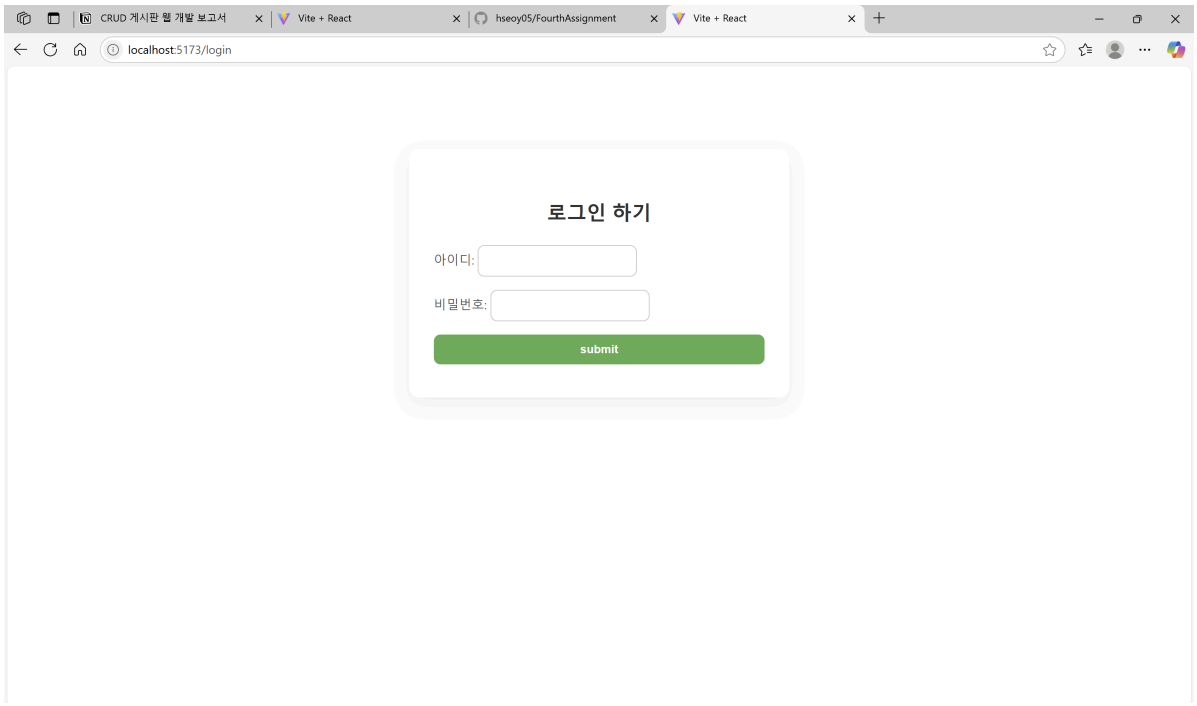
📝 CRUD 게시판 웹 개발 보고서

▼ 실제 돌아가는 화면

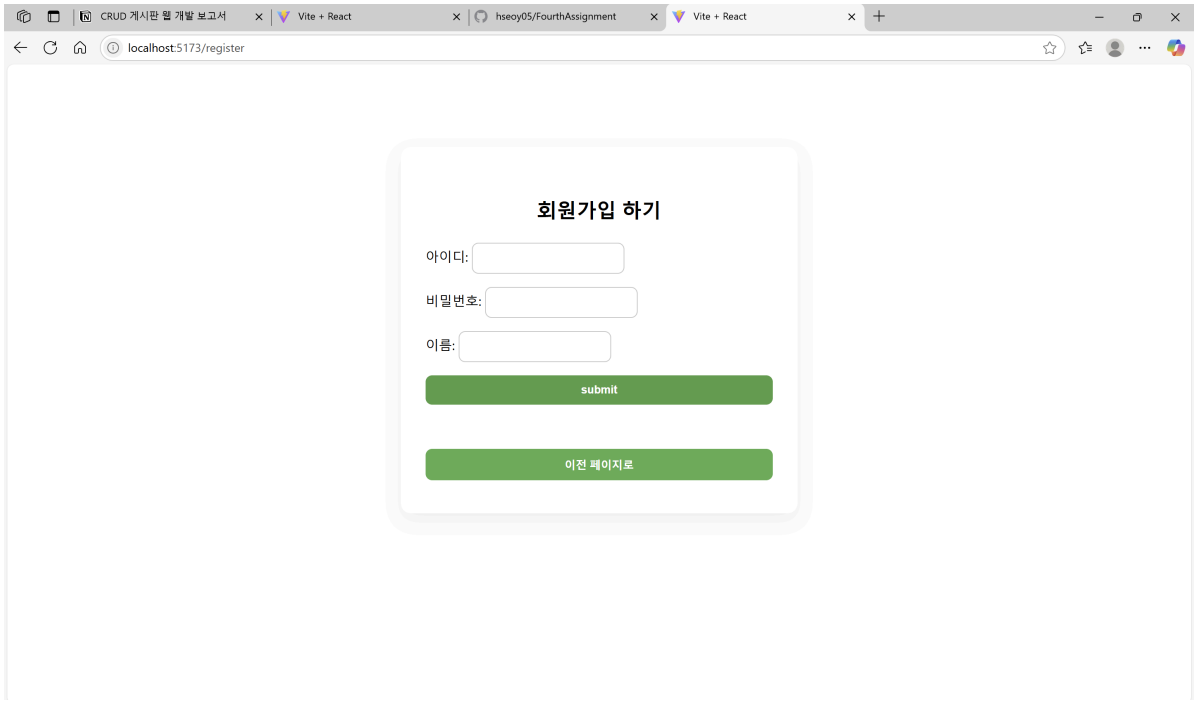
Home.jsx



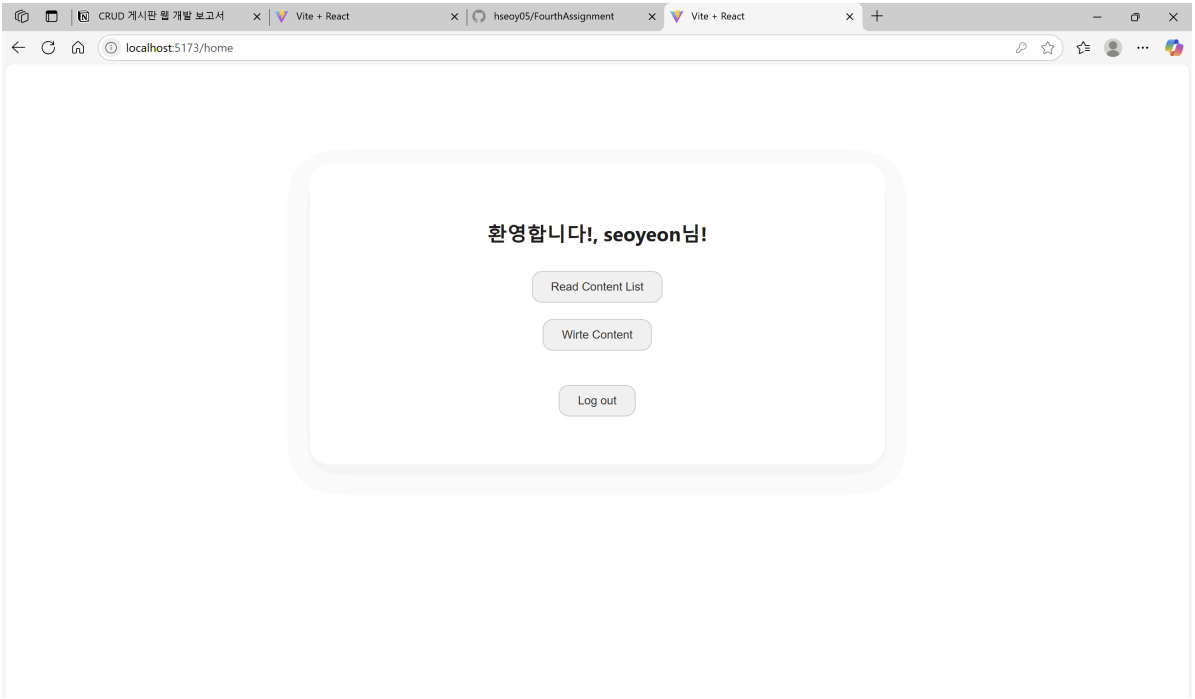
LoginPage.jsx



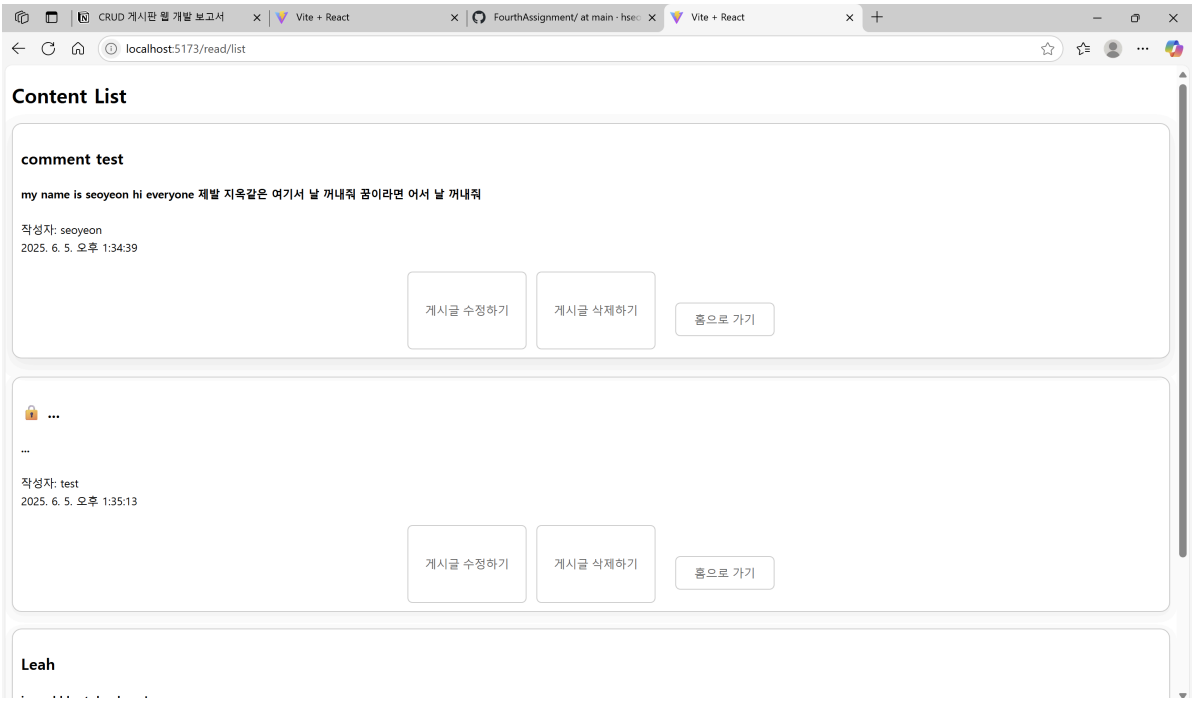
JoinPage.jsx



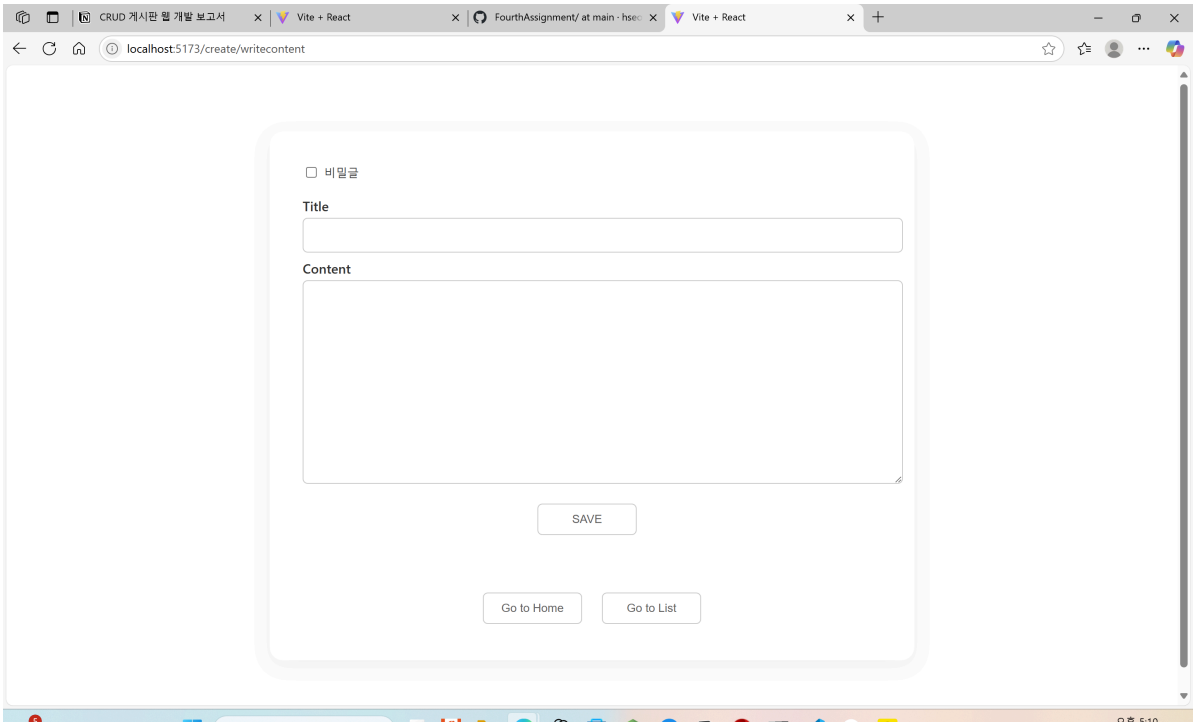
Home2.jsx



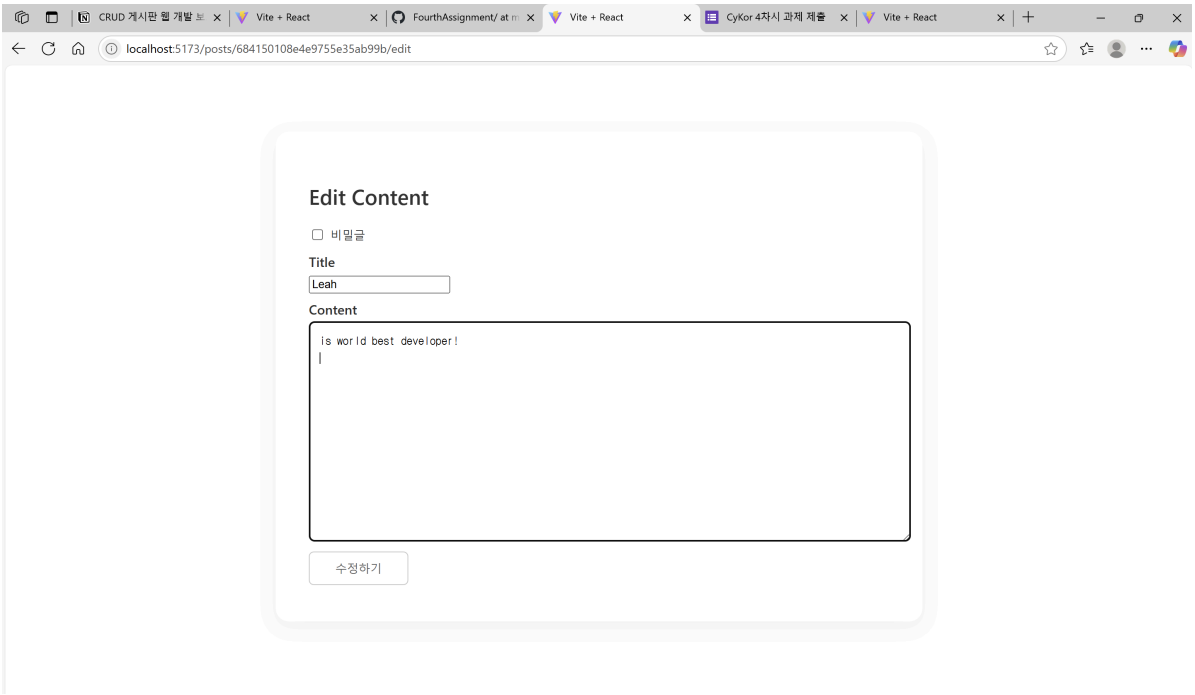
List.jsx



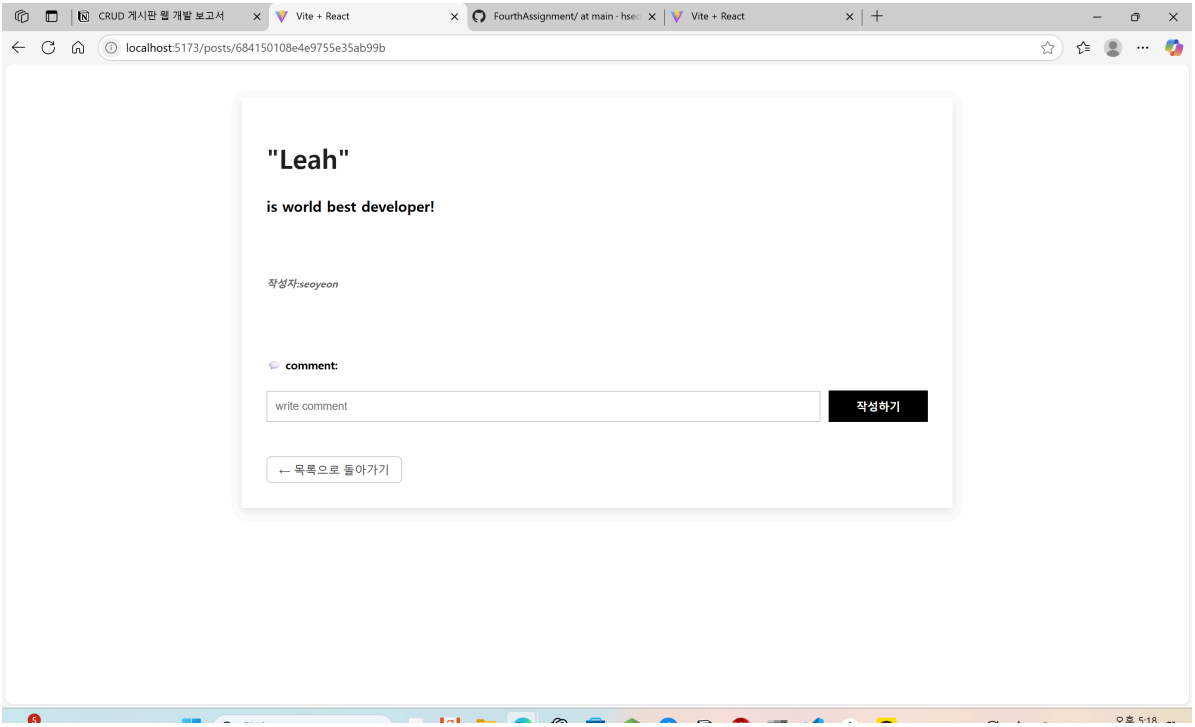
WriteConnect.jsx

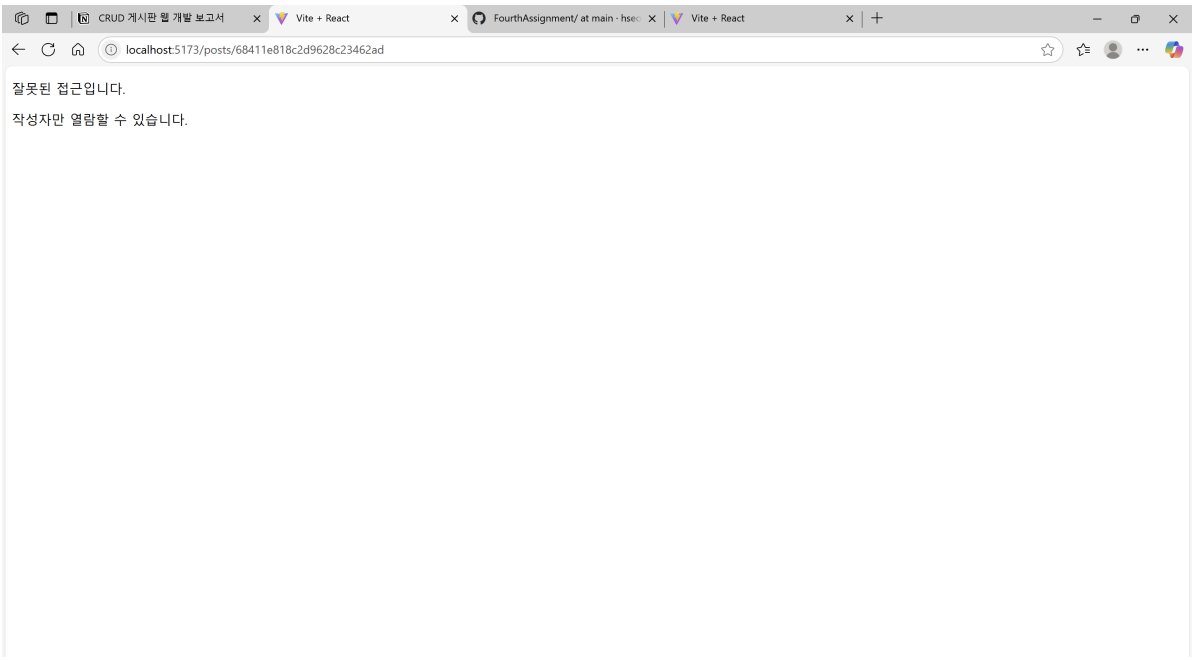


EditContent.jsx

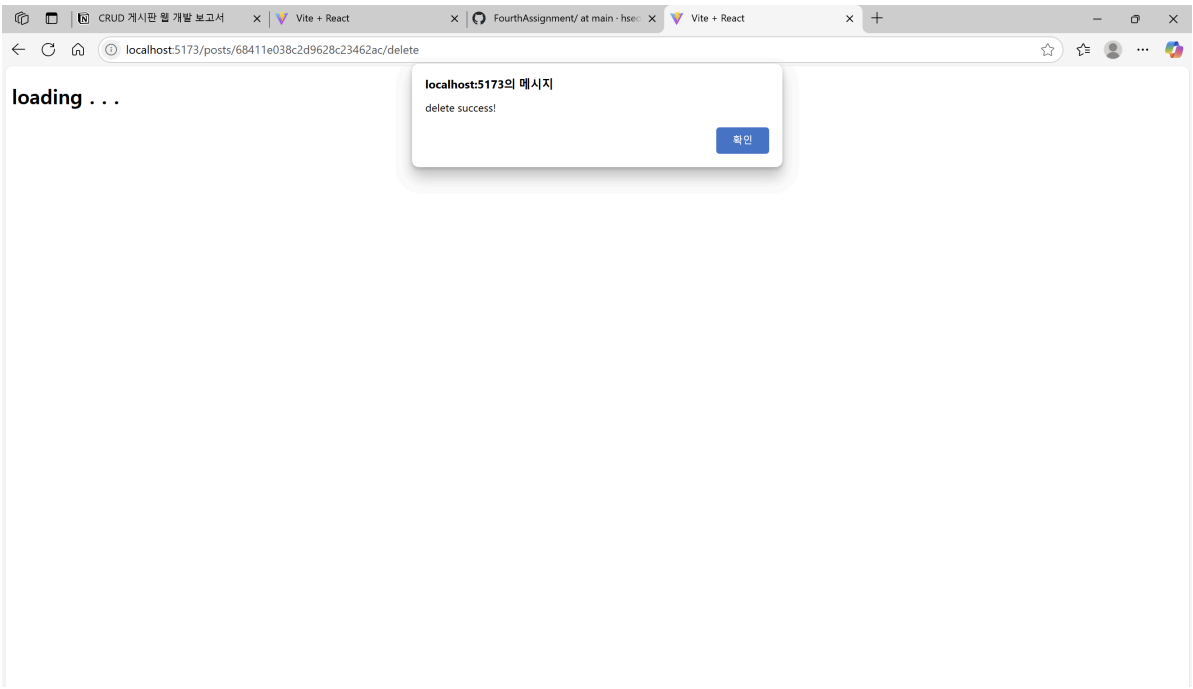


PostDetail.jsx



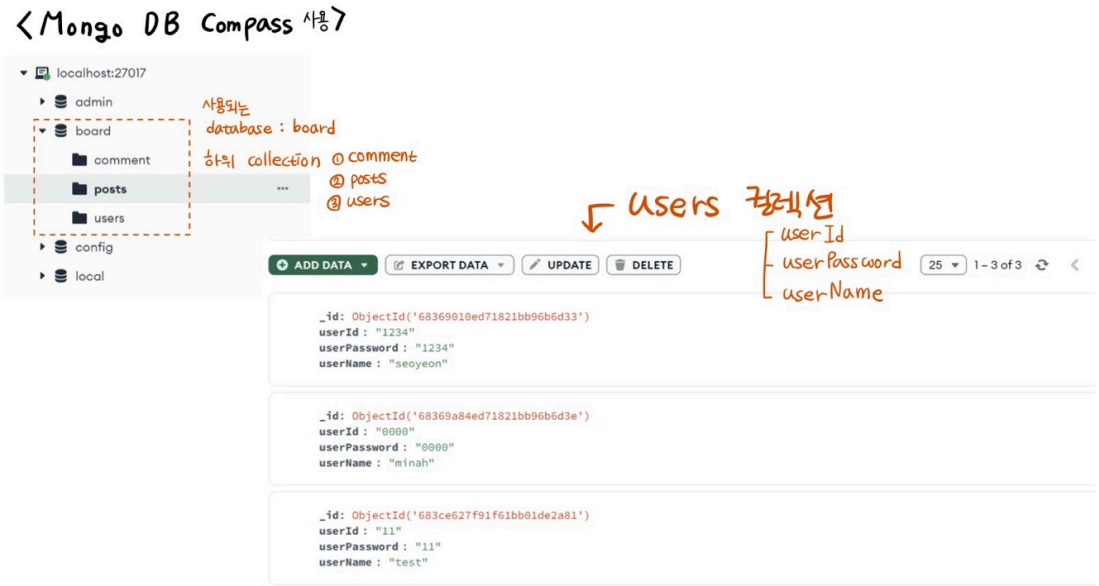


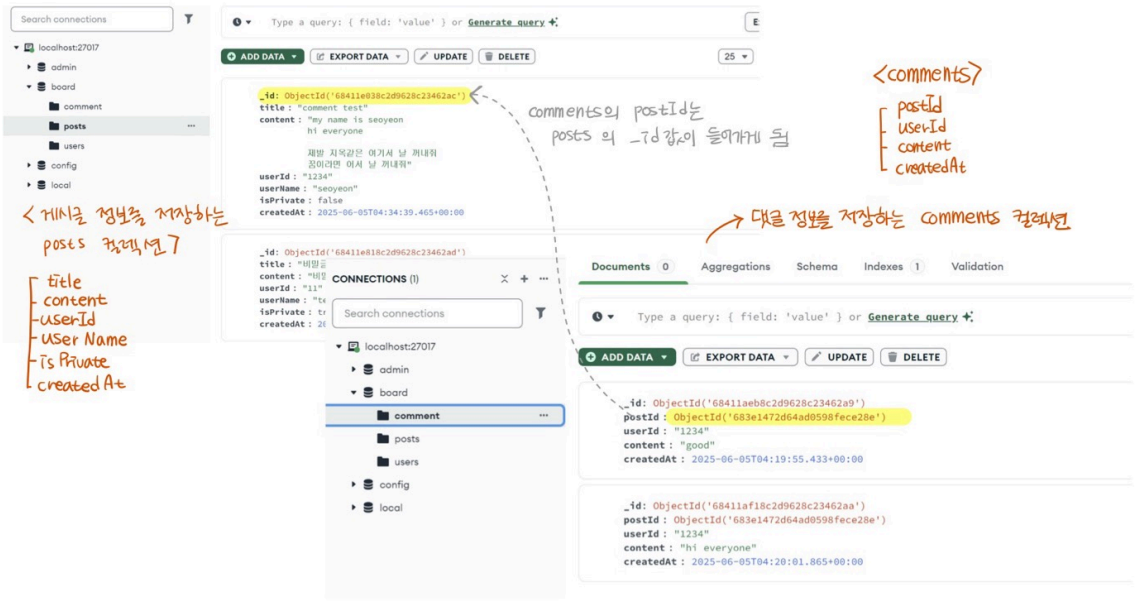
DeleteContent.jsx



▼ MongoDB

MongoDB Compass 사용했습니다.





[코드 개요]

1. 주요 기능

- 회원가입 / 로그인 (localStorage를 기반으로 로그인 유지)
- 게시글 작성 / 수정 / 삭제 / 목록 / 상세보기 / 게시글 댓글 달기
- 비밀글 설정 (🔐)

2. 프로젝트 구조도

```bash
Crud/
├── backend/          # 백엔드 (Express + MongoDB)
│   ├── models/       # Mongoose 모델 정의 (Comment)
│   ├── routes/       # API 라우터 (posts, comments, users)
│   ├── db.js         # MongoDB 연결 로직
│   ├── index.js      # 서버 엔트리포인트
│   └── Dockerfile    # 백엔드용 도커파일
├── frontend/
│   └── vite-project/ # 프론트엔드 (React + Vite)
│       ├── components/
│       ├── src/
│       ├── nginx.conf      # 정적서버 설정
│       └── Dockerfile      # 프론트엔드용 도커파일
├── docker-compose.yml      # 백/프런트 통합 배포 설정
```

3. 주요 기능 및 설명

게시글 (posts)

- 작성자(userId), 제목(title), 내용(content), 글을 쓴 시간(createdAt), 비밀글 여부
- 게시글 열람 시, 비밀글은 작성자 외에는 제목·내용이 '...' 으로 처리됨
- 수정/삭제는 작성자 자신만 가능하게끔 구현

댓글 (comments)

- userId를 localStorage를 활용해 가져와서, 단 사람의 아이디가 보이게끔 함
- postId라는 이름으로, 해당 게시글의 _id를 저장해서 게시글과 댓글을 연결함
- 댓글 추가, 불러오기 기능 구현

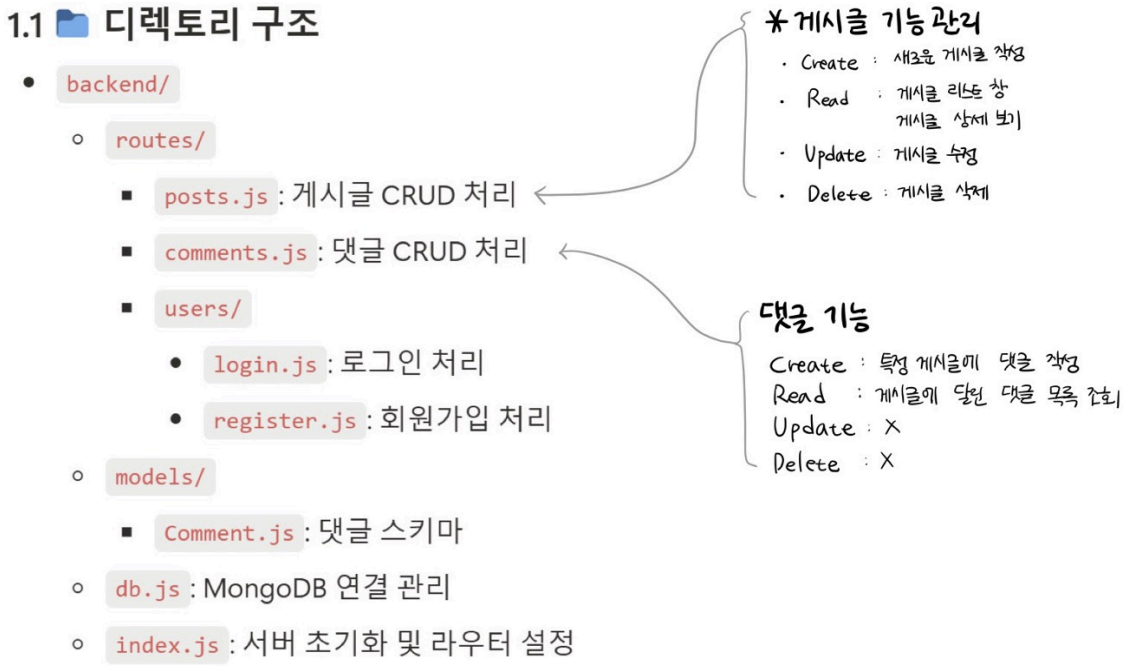
Login/Join

- 회원가입 (register) 시 userId, userPassword, userName 등록
- 로그인 시 userId를 localSotrage에 저장하여 DB에 그때끄때 저장할 수 있도록 함

[BACKEND]

backend/파일은 백엔드의 전체적인 파일들을 담은 상위 디렉토리는

주요 기능은 사용자 인증, 게시글 관리, 댓글 기능 등입니다.



*제출 후 시간 여유가 생겨 댓글 기능 Delete 까지 추가했습니다!

[FRONTEND]

영상에서 했던 것처럼, Vite 기반의 React 파일로 개발하였습니다. 하나의 html 페이지 내에서 필요한 부분만 동적으로 바뀌게끔 코딩하였습니다.

페이지 간 이동과 URL 파라미터 처리는 **'React Router DOM'** 속 **useNavigate**를 주로 사용하였습니다.

서버와의 데이터 송수신은 **Axios**를 사용했습니다.

사용자의 로그인 상태는 **LocalStorage**를 활용했습니다. user정보 중 userPassword와 userName은 저장하지 않고, **userId**만 저장했습니다.

css 는 따로 component라는 파일을 만들어 이 안에 다 정리를 해 놓았습니다. (Ai 활용)

1. 디렉토리 구조

```
frontend/vite-project/
├── component/            → css 파일 모음
├── App.jsx               → 전체 라우팅 설정
├── src/
│   ├── Home.jsx          → 사이트 접속 시 나오는 첫 화면
│   ├── Home2.jsx         → 로그인 후 나오는 첫 화면
│   └── pages/
│       ├── Create/WriteContent.jsx      → 게시글 쓰기
│       ├── Update/EditContent.jsx       → 게시글 수정
│       ├── Read/List.jsx                → 게시글 목록
│       ├── Read/PageDetail.jsx          → 게시글 상세보기
│       └── Delete/DeleteContent.jsx     → 게시글 삭제
│   ├── loginOut/
│       ├── loginPage.jsx        → 로그인 화면
│       └── joinPage.jsx         → 회원가입 화면
└── main.jsx → 루트 지정 파일
```

2. 전체 동작 흐름 요약 (Data Flow)

loginOut/

- 사용자 정보 입력 → POST 요청 → MongoDB 저장
- 로그인 성공 시 LocalStorage에 `userId` 저장
  → 추후 게시글 수정, 삭제, 비밀글에 사용됨

posts/Create

- 제목, 내용, `비밀글 여부` 입력 → POST `/posts`

posts/Read

- GET `/posts` → 전체 목록 출력
- 비밀글이면 제목, 내용 숨김 표시

posts/Update, Delete

- 사용자 인증 후 `PUT` or `DELETE` 요청

[추가로 구현한 기능들(1)]

1. 🔐 비밀글 기능 구현

작성 시 체크박스 처리 (in WriteContent.jsx)

비밀글 여부를 `boolean` 타입으로 백엔드에 전달

```
const [isPrivate, setIsPrivate] = useState(false);
<input type="checkbox" onChange={() => setIsPrivate(!isPrivate)} />
```

작성 시 `POST /posts` 요청에 `isPrivate` 필드 포함

```
body: JSON.stringify({ title, content, userId, isPrivate }),
```

MongoDB에 저장 (in posts.js - post 라우터)

```
const result = await db.collection('posts').insertOne({
  title, content, userId, userName, isPrivate: !!isPrivate, createdAt: new Date()
});
```

목록에서 필터링 없이 전체 노출 (in List.jsx)

```
{posts.map(post => (
  <div key={post._id}>
    {post.isPrivate ? "🔒 비밀글입니다" : post.title}
  </div>
))}
```

if 작성자가 아닐 시

즉, 게시글 속 userId와 현재 localStorage에 저장된 userId가 다를 시에

상세보기 접근 제한 (PostDetail.jsx)

```
useEffect(() => {
  if (post.isPrivate && post.userId !== localStorage.getItem("userId")) {
    alert("비밀글은 작성자만 볼 수 있습니다.");
    navigate("/read/list");
  }
}, [post]);
```

[추가로 구현한 기능들(2)]

2. 📝 댓글 기능 구현 상세 분석

댓글 스키마 정의 (in `models/Comment.js` **)**

```
const commentSchema = new mongoose.Schema({
  postId: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'Post'
},
  userId: { type: String, required: true },
  content: { type: String, required: true },
```

```
  createdAt: { type: Date, default: Date.now },
});
```

각댓글은 특정 게시글 postId(게시글 데이터 테이블에서는 _id 역할을 하는 값)에 연결되며, 사용자 ID(userId), 댓글 내용(content), 생성일(createdAt)을 저장함.

댓글 API 라우터 (in routes/comments.js)

```
router.get('/:postId', async (req, res) => {
  const comments = await Comment.find({ postId: new ObjectId(req.params.p
ostId) }).sort({ createdAt: -1 });
  res.json(comments);
});

router.post('/', async (req, res) => {
  const newComment = new Comment(req.body);
  const savedComment = await newComment.save();
  res.status(200).json(savedComment);
});
```

`GET /comments/:postId` : 특정 게시글의 댓글 목록을 최신순으로 조회.

`POST /comments` : 댓글 데이터를 받아 DB에 저장 후 응답으로 반환.

그리고 프론트로 넘어와서 PostDetail 안에서 댓글 코드를 작업함.

In PostDetail.jsx ...

useState를 활용해 comments 배열을 선언하여 메인으로 활용했고, Ref도 이용함

*Ref 를 사용한 이유:
댓글 입력창에 자동으로 커서를 다시 옮겨주기 위함 → 사용자가 바로 다음 댓글을 편하게 쓸 수 있도록 하는 기능을 넣고싶었음

댓글 목록 불러오기:

```
const fetchComments = async () => {
  const res = await fetch(`http://localhost:8800/comments/${id}`);
  const data = await res.json();
  setComments(Array.isArray(data) ? data : []);
};
```

댓글 등록 함수:

```
const handleCommentSubmit = async () => {
  const res = await fetch('http://localhost:8800/comments', {
    method: 'POST',
    headers: { 'Content-type': 'application/json' },
    body: JSON.stringify({
      postId: id,
      userId: localStorage.getItem("userId"),
      content: commentInput,
    })
  });
  await fetchComments(); // 새 댓글 반영
  setCommentInput('');
  commentInputRef.current?.focus();
};
```

댓글 입력 및 출력 UI:

```
<input
  type="text"
  value={commentInput}
  onChange={(e) => setCommentInput(e.target.value)}
  ref={commentInputRef}
/>
<button onClick={handleCommentSubmit}>작성하기</button>

<ul>
  {comments.map((c) => (
```

```
    <li key={c._id}><strong>{c.userId}</strong>: {c.content}</li>
  ))}
</ul>
```

댓글을 입력하고 작성하기 버튼을 누르면서버에 POST 요청

→ 성공 시 댓글 목록을 다시 불러와 렌더링.

댓글은 postId(_id)로 연결되며, 해당 게시글에만 종속됨.

localStorage에 저장된 userId를 통해 댓글 작성자 이름을 같이 기록함.

UI 입력 → API 요청 → DB 저장 → 목록 갱신

Dockerfile

`crud-backend` : localhost:8800

`crud-frontend` : localhost:5147

`mongo` : 내부 네트워크에서 접속, 포트 27017

모든 컨테이너는 Docker volume을 사용하여, 로컬 파일 시스템과 컨테이너의 파일 시스템을 연결함. 이를 통해 로컬에서의 파일 수정 사항이 컨테이너 내부에 실시간으로 반영되도록 구성

컨테이너를 재빌드하거나 재시작하지 않고도 소스 코드를 즉시 테스트할 수 있는,,,

효율적인 개발 환경을 맞춰놓고 코드를 짰음