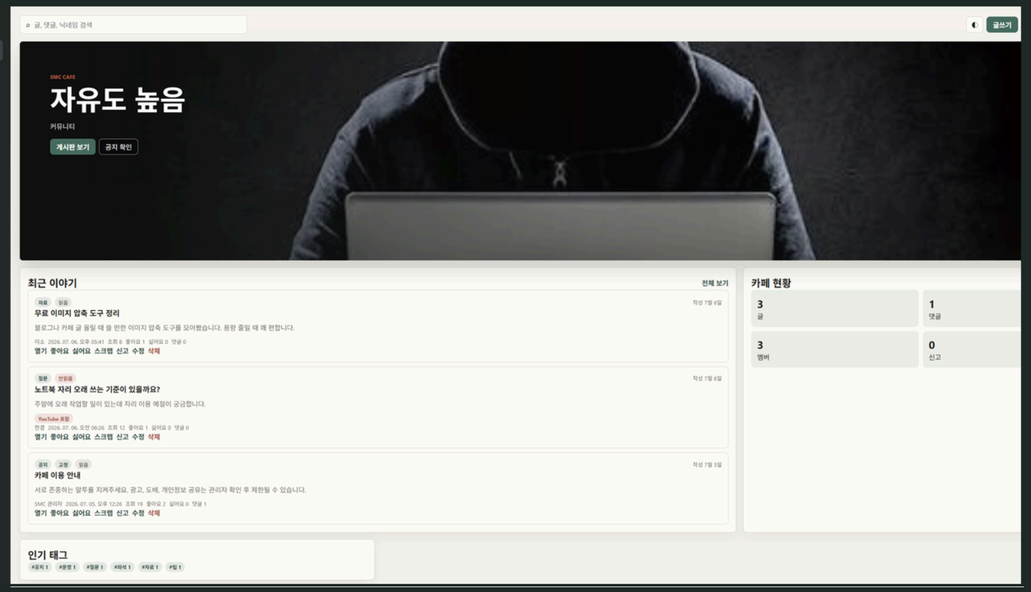
SMC Cafe

인터넷 카페 코드 분석 발표

HTML, CSS, JavaScript, JSON, Node.js가 어디에 쓰였는지 실제 코드와 실행 화면으로 정리한 자료임

분석 기준: 업로드한 smc-cafe 프로젝트 파일을 열어 구조를 확인하고, 서버 실행 후 화면 동작까지 확인함



차례

기능별로 코드가 어떤 역할을 하는지 따라가는 방식임

직접 실행 확인

터미널에서 서버를 켜고 웹페이지가 열리는지 확인한 기록임

```
PS C:\Users\zhang\Desktop\카페\smc-cafe>
node server.js

서버 실행 중: http://localhost:3000

브라우저 확인 화면
홈 화면, 게시판, 로그인, 글쓰기 저장, 관리자 화면 확인함

확인 결과
Express 서버가 public 폴더를 웹사이트로 열고 index.html을 불러오는 구조임
```

실행해서 확인한 흐름

1. smc-cafe 폴더에서 node server.js를 실행함

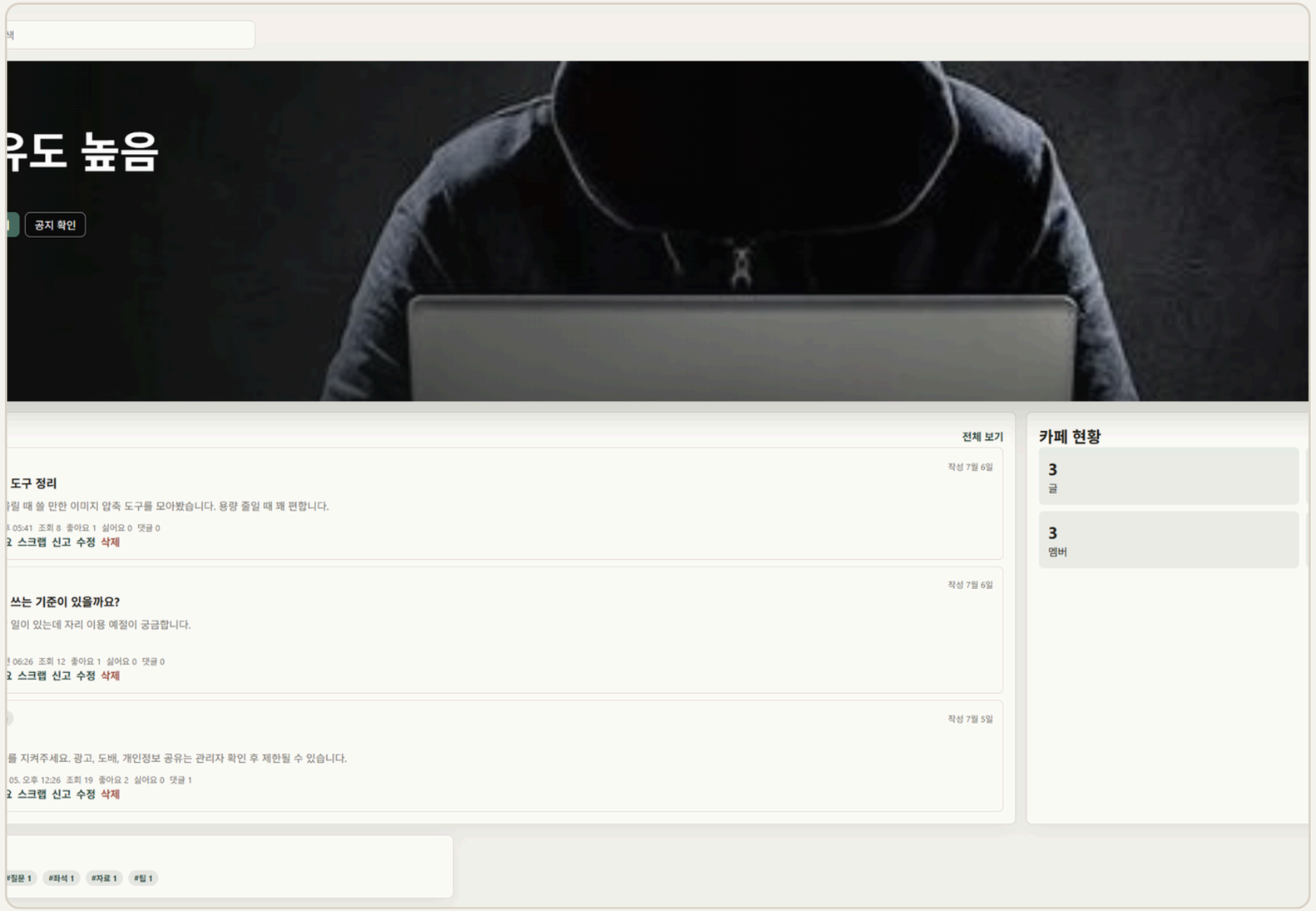
2. 터미널에 서버 실행 중: http://localhost:3000 문구가 출력됨

3. public 폴더의 index.html이 웹사이트 첫 화면으로 열리는 구조임

4. 로그인, 글쓰기, 관리자 화면까지 화면 동작을 눌러 확인함

실행 화면 1

홈 화면을 열었을 때 보이는 첫 화면임



화면에서 확인한 부분

왼쪽에는 사이드바가 있고, 위쪽에는 검색창과 로그인 버튼이 있음

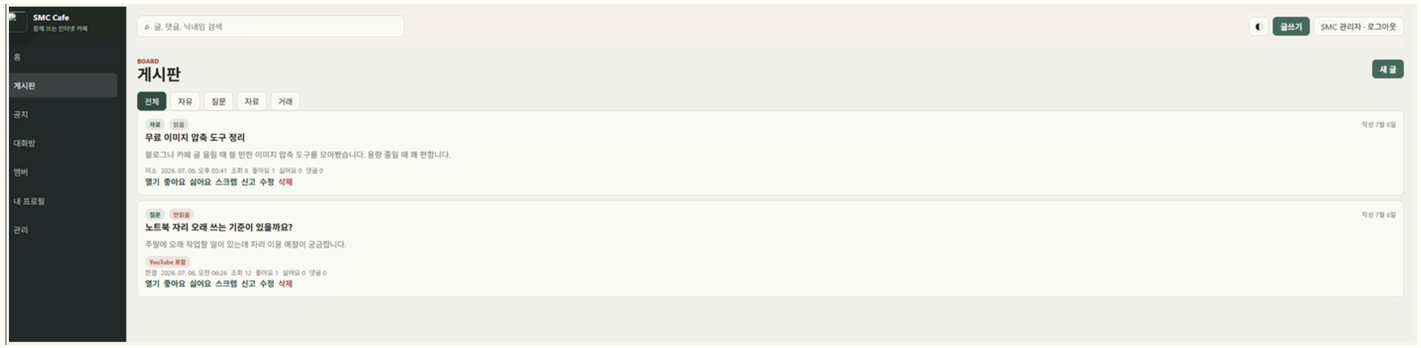
가운데 큰 영역은 hero 섹션임

아래에는 최근 이야기, 카페 현황, 인기 태그가 카드 형태로 배치됨

HTML 구조와 CSS grid가 같이 작동한 결과임

실행 화면 2

게시판 메뉴를 눌렀을 때 보이는 화면임



화면에서 확인한 부분

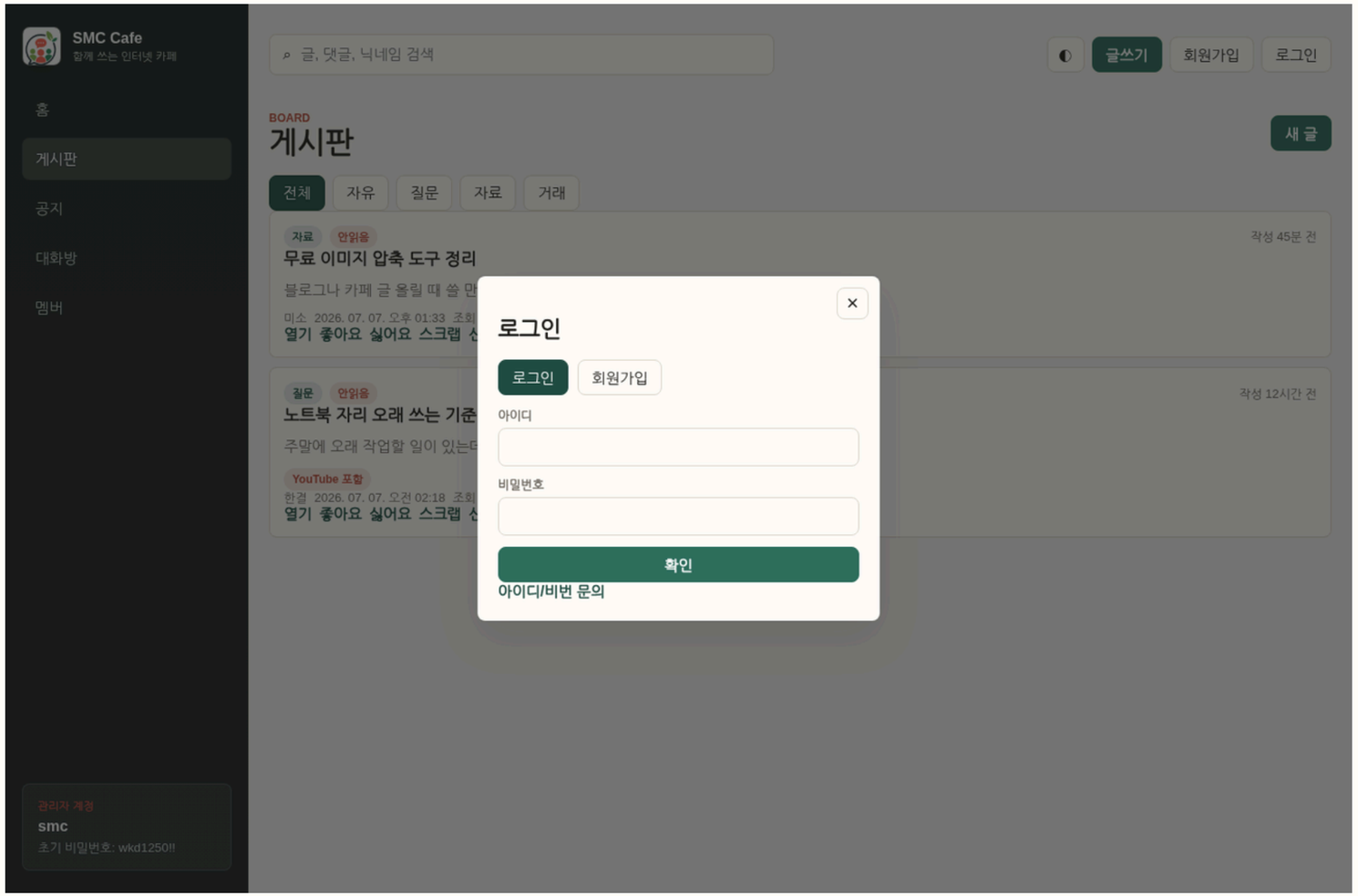
카테고리 chip 버튼으로 전체, 자유, 질문, 자료, 거래를 나눔

글 카드에는 작성자, 시간, 조회수, 좋아요, 싫어요, 댓글 수가 같이 표시됨

이 화면은 renderBoard와 renderPostCard가 만든 결과임

실행 화면 3

로그인 모달을 직접 열어 본 화면임



화면에서 확인한 부분

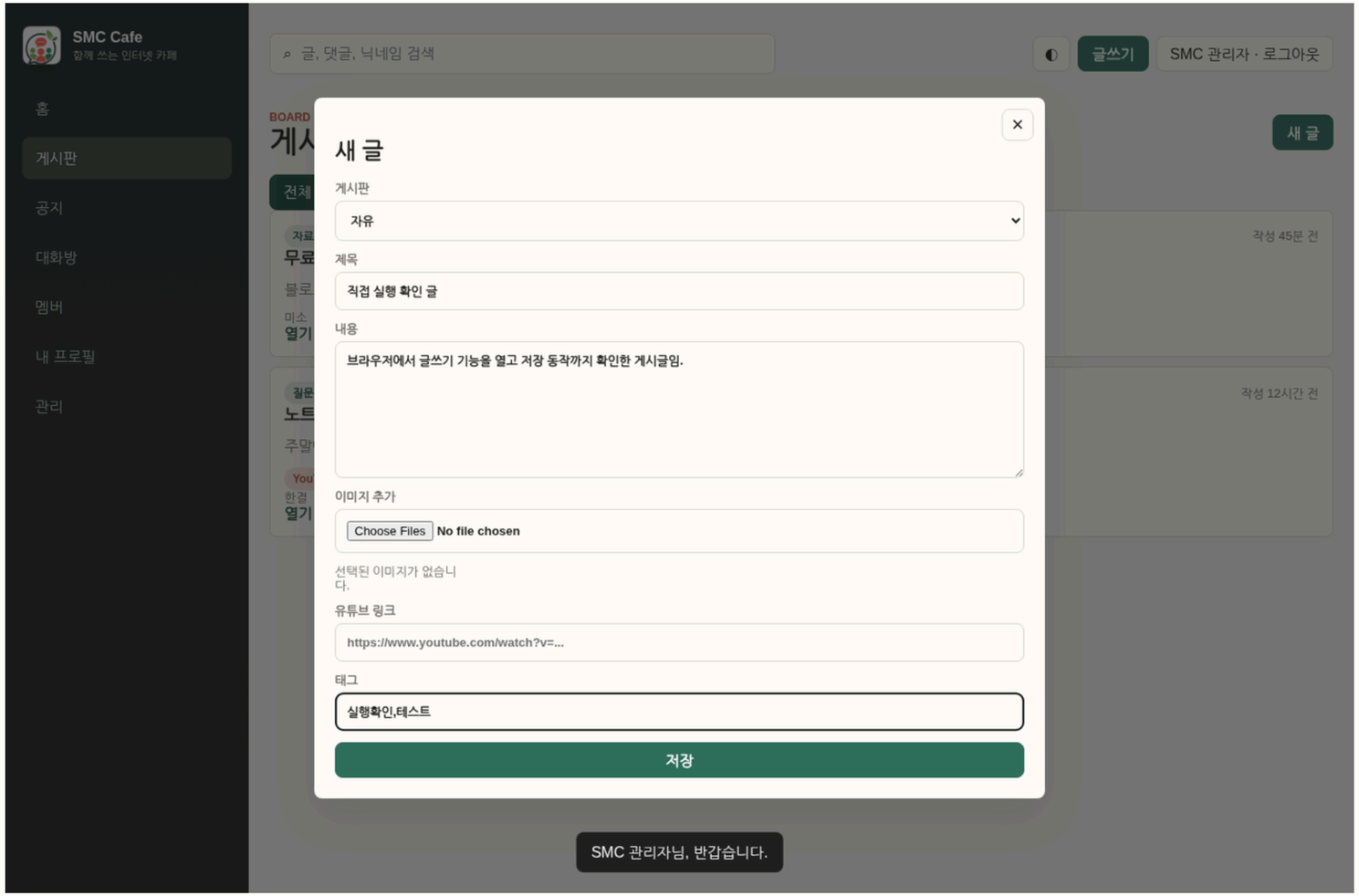
로그인과 회원가입을 같은 모달에서 처리함

authMode 값이 login인지 join인지에 따라 입력칸과 제목이 바뀌는 구조임

확인 버튼을 누르면 authForm submit 이벤트가 실행됨

실행 화면 4

글쓰기 창에 값을 넣고 저장하기 직전 화면임



화면에서 확인한 부분

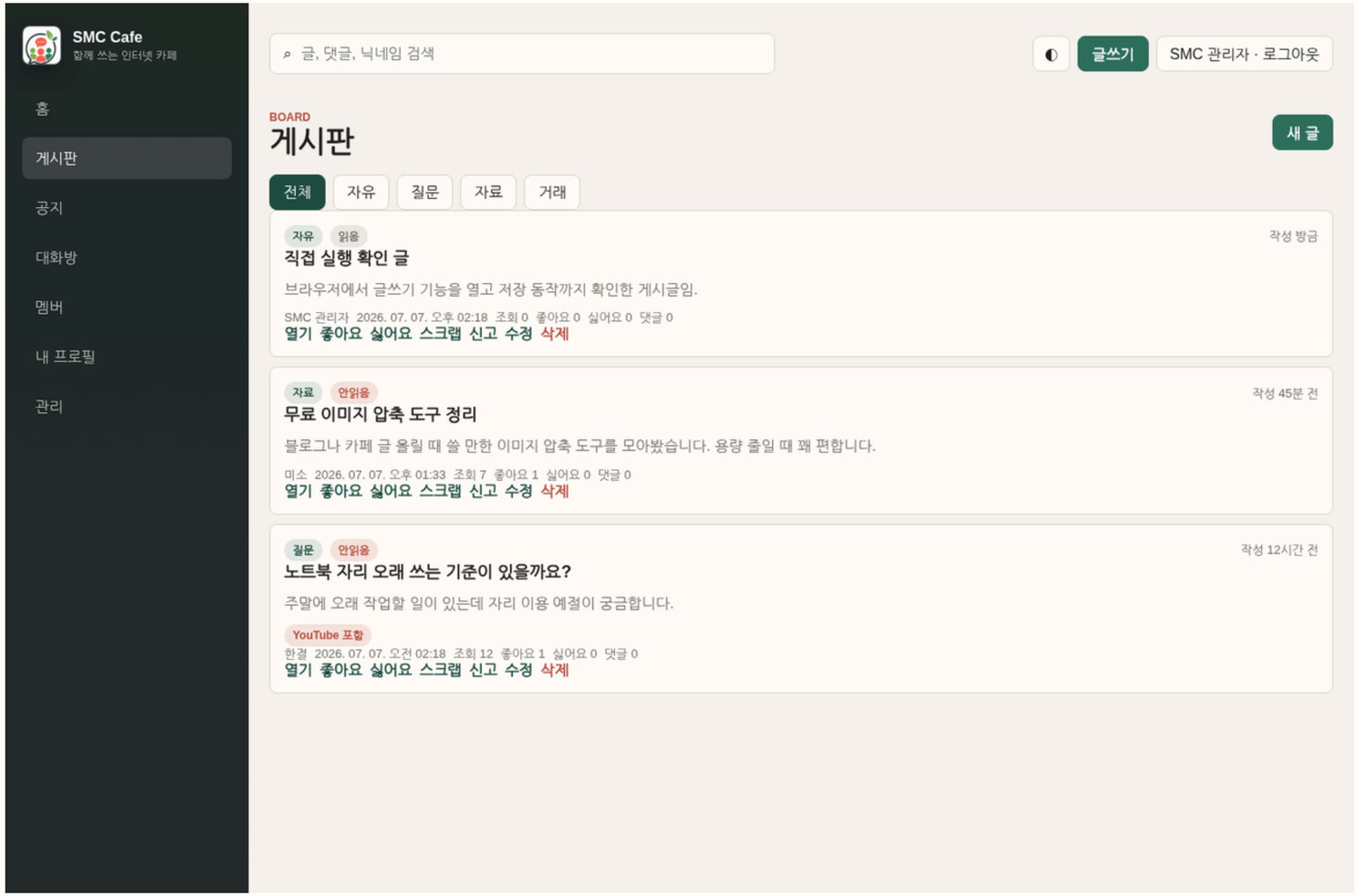
게시판 종류, 제목, 내용, 이미지, 유튜브 링크, 태그를 한 번에 입력하는 폼임

저장 버튼을 누르면 postForm submit 이벤트가 실행됨

새 글은 state.posts 맨 앞에 추가되어 최신순으로 보이게 됨

실행 화면 5

글쓰기 저장 뒤 게시판에서 확인한 화면임



화면에서 확인한 부분

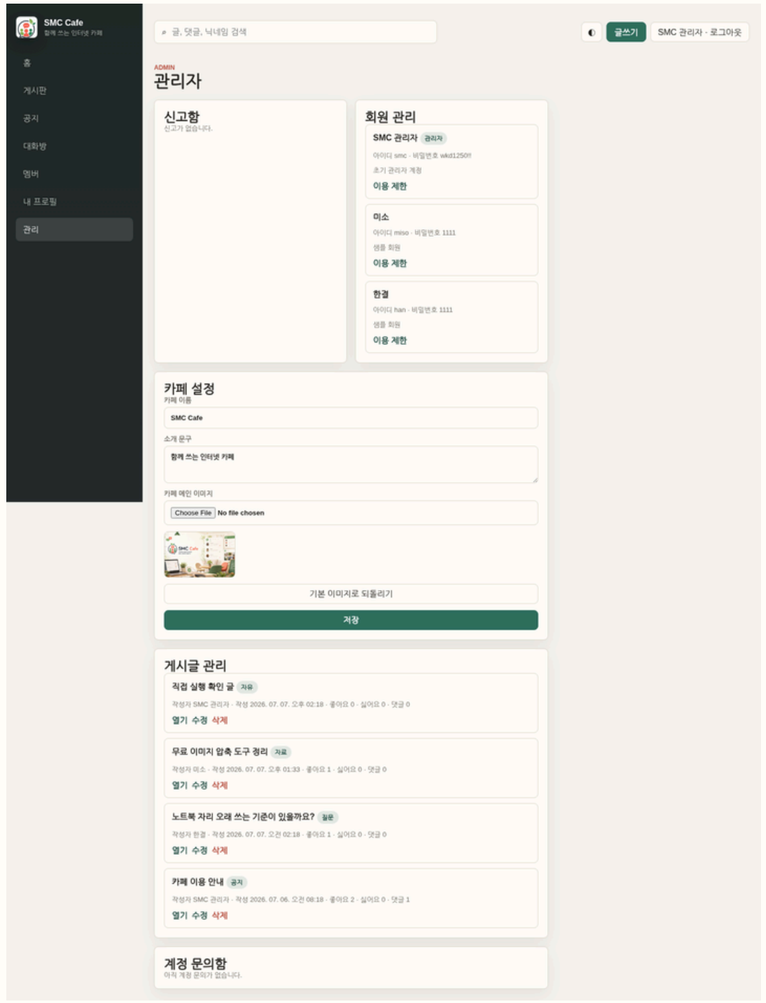
직접 작성한 테스트 글이 게시판 맨 위에 올라온 상태임

이 동작은 posts.unshift로 새 글을 앞쪽에 넣기 때문에 가능함

저장 뒤 render 함수가 다시 실행되어 화면이 바로 갱신됨

실행 화면 6

관리자 계정으로 들어간 관리자 화면임



화면에서 확인한 부분

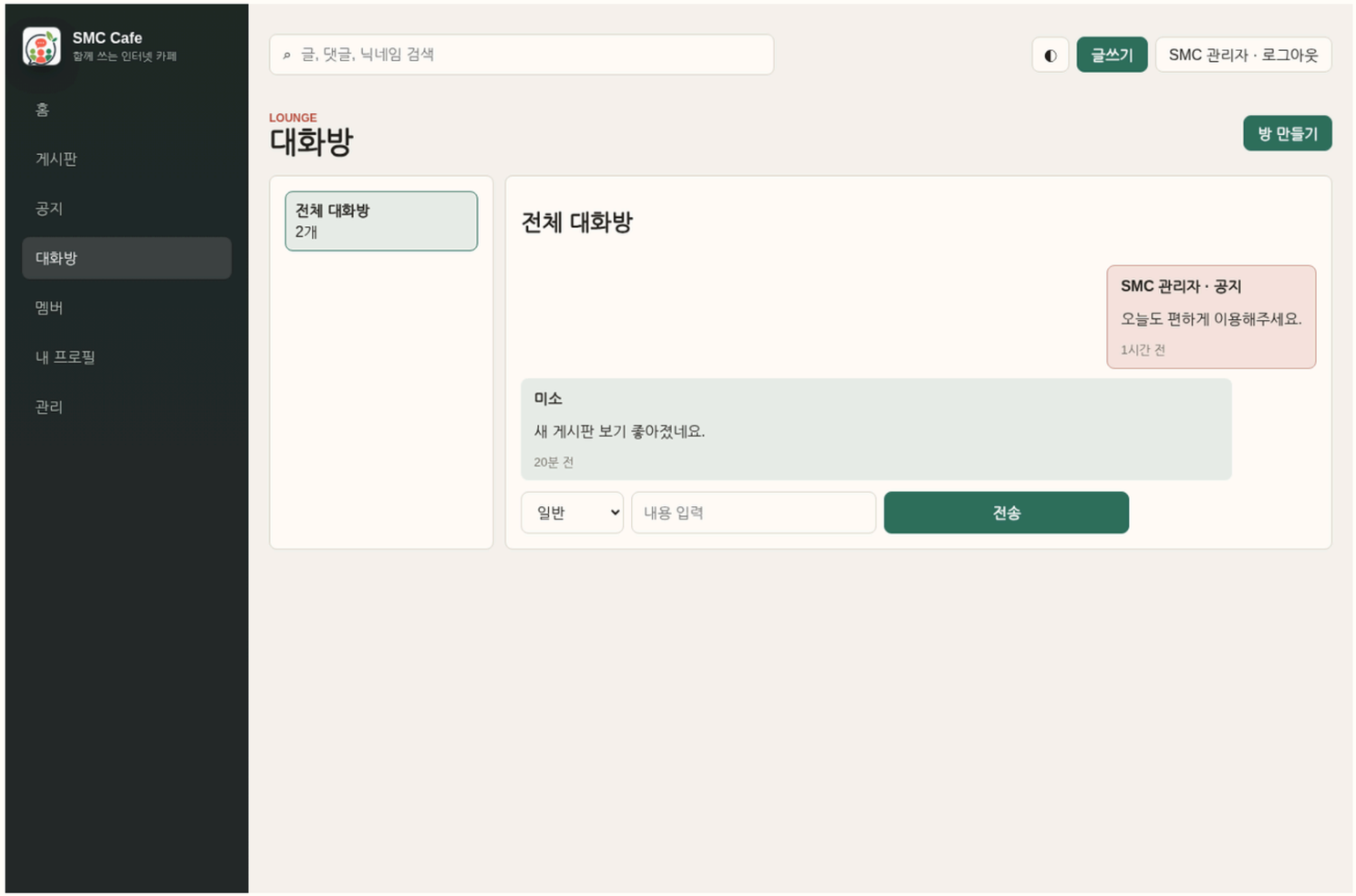
관리자 화면에는 신고함, 회원 관리, 카페 설정, 게시글 관리, 계정 문의함이 있음

관리자만 보이도록 admin-only 클래스와 isAdmin 함수가 같이 쓰임

일반 회원에게는 이 메뉴가 숨겨지는 구조임

실행 화면 7

대화방 화면을 열었을 때 보이는 구조임



화면에서 확인한 부분

왼쪽에는 대화방 목록이 있고 오른쪽에는 메시지 로그가 있음

일반 메시지, 공지, 투표를 선택해서 보낼 수 있음

chatRooms 배열 안에 방과 메시지를 같이 저장하는 방식임

프로젝트 파일 구조

각 파일이 맡은 역할을 먼저 잡아야 코드가 이해됨

```
smc-cafe
[  public
   index.html
   styles.css
   app.js]
 assets
   cafe-hero-generated.png
   smc-logo-mark.png
 [data
   posts.json]
 server.js
 package.json
 Dockerfile
 netlify.toml
 vercel.json
```

index.html은 화면의 뼈대를 만드는 파일임

styles.css는 색상, 간격, 카드 모양, 반응형 화면을 담당함

app.js는 로그인, 게시글, 댓글, 관리자, 채팅 같은 실제 기능을 움직이는 파일임

server.js는 Express 서버로 public 폴더를 웹사이트로 열어주는 파일임

posts.json은 서버형 게시글 저장을 위해 준비된 JSON 파일임

사용한 기술 정리

이 프로젝트에 들어간 기술을 역할 기준으로 나눠 본 내용임

1	**HTML**	브라우저에 보여질 제목, 버튼, 입력창, 모달, 게시판 영역 같은 구조를 만드는 언어임
2	**CSS**	색상, 글자 크기, 배치, 카드 모양, 다크모드 같은 디자인을 만드는 언어임
3	**JavaScript**	사용자가 버튼을 누르면 화면을 바꾸고 데이터를 저장하게 만드는 프로그래밍 언어임
4	**JSON**	게시글, 회원, 댓글 같은 데이터를 객체와 배열 형태로 저장하기 위한 데이터 형식임
5	**Node.js**	브라우저 밖에서 JavaScript를 실행해서 서버를 만들 수 있게 해주는 실행 환경임
6	**Express**	Node.js에서 웹서버와 API 주소를 쉽게 만들기 위해 쓰는 서버 라이브러리임

HTML 기본 설정

문서 인코딩과 모바일 화면 기준을 잡는 부분임

실제 코드

```
<!doctype html>
<html lang="ko">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-
width, initial-scale=1.0" />
    <title>SMC Cafe</title>
    <link rel="stylesheet" href="./styles.css" />
  </head>
```

어떻게 쓰는 코드인지

charset을 UTF-8로 지정해서 한글이 깨지지 않게 함

viewport는 휴대폰이나 태블릿에서도 화면 크기에 맞게 보이게 만드는 설정임

styles.css를 연결해서 HTML 구조에 디자인이 적용되도록 함

왜 이렇게 사용하는지

HTML은 글자와 버튼을 화면에 놓는 뼈대임

CSS와 JS를 연결해야 디자인과 기능이 붙기 때문에 head에서 기본 연결을 먼저 해두는 구조임

전체 화면 뼈대

사이드바와 메인 화면을 나누는 구조임

실제 코드

```
<body>
  <div class="app-shell">
    <aside class="sidebar">
      ... 메뉴 영역 ...
    </aside>

    <main class="main">
      ... 실제 화면 영역 ...
    </main>
  </div>
</body>
```

어떻게 쓰는 코드인지

app-shell은 사이트 전체를 감싸는 가장 큰 박스임

aside는 왼쪽 메뉴처럼 보이는 영역이고 main은 홈, 게시판, 관리자 같은 실제 콘텐츠가 들어가는 영역임

왜 이렇게 사용하는지

화면을 처음부터 사이드바와 본문으로 나눠두면 나중에 게시판, 공지, 대화방 같은 페이지를 바꿔도 기본 틀은 그대로 유지할 수 있음

메뉴 이동 버튼

data-route로 어느 화면으로 갈지 표시하는 방식임

실제 코드

```
<nav class="nav-list" aria-label="주요 메뉴">
  <button class="nav-item active" data-route="home">홈</button>
  <button class="nav-item" data-route="board">게시판</button>
  <button class="nav-item" data-route="notice">공지</button>
  <button class="nav-item" data-route="chat">대화방</button>
  <button class="nav-item admin-only" data-route="admin">관리</button>
</nav>
```

어떻게 쓰는 코드인지

버튼마다 data-route 값이 들어 있음

JS는 사용자가 어떤 버튼을 눌렀는지 보고 그 값에 맞는 화면만 보여줌

admin-only는 관리자에게만 보이는 메뉴를 표시하기 위한 클래스임

왜 이렇게 사용하는지

페이지를 여러 html 파일로 쪼개지 않고 한 화면 안에서 섹션만 바꾸는 방식임

그래서 이동 속도가 빠르고, JS 상태값을 유지하기 쉬운 구조임

홈 화면 섹션

처음 접속했을 때 보이는 영역을 HTML로 미리 만들어 둔 부분임

실제 코드

```
<section id="homeView" class="view active-view">
  <div class="hero">
    <img id="heroImage" src="./assets/cafe-hero-generated.png" />
    <div class="hero-copy">
      <h1>동네 카페처럼 편하게 모이는 공간</h1>
      <button data-route="board">게시판 보기</button>
    </div>
  </div>
</section>
```

어떻게 쓰는 코드인지

homeView는 홈 화면을 뜻함

active-view가 붙어 있기 때문에 처음에는 홈 화면이 보임

heroImage와 hero-copy는 메인 이미지와 소개 문구를 담당함

왜 이렇게 사용하는지

처음부터 홈 영역을 만들어두고 다른 화면은 display none으로 숨기는 구조임

JS가 class만 바꾸면 화면 전환이 되기 때문에 구현이 단순해짐

게시판 화면 HTML

게시글 목록이 들어갈 자리를 미리 만들어 둔 구조임

실제 코드

```
<section id="boardView" class="view">
  <div class="filter-row">
    <button class="chip active" data-category="all">전체
</button>
    <button class="chip" data-category="free">자유
</button>
    <button class="chip" data-category="question">질문
</button>
  </div>
  <div id="boardList" class="post-stack"></div>
</section>
```

어떻게 쓰는 코드인지

boardList는 처음에는 비어 있는 div임

JS가 게시글 데이터를 읽은 뒤 HTML 문자열을 만들어 이 안에 넣음

chip 버튼은 게시판 카테고리 필터 역할을 함

왜 이렇게 사용하는지

게시글 수가 계속 바뀌기 때문에 HTML에 글을 직접 적지 않음

데이터를 기준으로 JS가 목록을 다시 만드는 방식이 훨씬 관리하기 쉬움

모달 HTML

로그인, 글쓰기, 상세보기 창을 팝업처럼 띄우는 구조임

실제 코드

```
<div class="modal" id="authModal" aria-
hidden="true">
  <div class="modal-card">
    <button class="modal-close" data-
close="authModal">×</button>
    <h2 id="authTitle">로그인</h2>
    <form id="authForm" class="form-stack">
      <input id="authId" required />
      <input id="authPw" type="password" required />
      <button type="submit">확인</button>
    </form>
  </div>
</div>
```

어떻게 쓰는 코드인지

modal은 평소에는 숨겨져 있다가 로그인 버튼을 누르면 열림

authForm 안의 입력값을 JS가 읽어서 로그인이나 회원가입을 처리함

왜 이렇게 사용하는지

모달을 쓰면 페이지를 새로 이동하지 않고도 로그인, 글쓰기, 상세보기를 같은 화면 위에서 처리할 수 있음

사용자가 흐름을 끊지 않고 작업할 수 있는 방식임

CSS 변수

색상을 한곳에 모아 관리하는 부분임

실제 코드

```
:root {
  --bg: #f5f3ee;
  --surface: #fffdf9;
  --ink: #23211d;
  --muted: #746f67;
  --brand: #2f6f5e;
  --accent: #b94f3f;
  --shadow: 0 18px 45px rgba(47, 44, 38, 0.12);
}
```

어떻게 쓰는 코드인지

CSS 변수는 같은 색상을 여러 곳에서 반복해서 쓸 때 편함

예를 들어 var(--brand)를 버튼, 링크, 강조 글자에 같이 적용할 수 있음

왜 이렇게 사용하는지

디자인을 바꿀 때 모든 버튼을 하나씩 수정하지 않아도 됨

색상표만 바꾸면 사이트 전체 분위기가 같이 바뀌는 구조임

다크 모드

data-theme 값으로 전체 색상을 바꾸는 방식임

실제 코드

```
[data-theme="dark"] {
  --bg: #171917;
  --surface: #22241f;
  --ink: #f1eee7;
  --muted: #b9b0a4;
  --brand: #73ae96;
  --accent: #d88472;
}
```

어떻게 쓰는 코드인지

body에 data-theme="dark"가 붙으면 CSS 변수가 어두운 색상으로 바뀜

버튼이나 카드가 직접 색을 바꾸는 게 아니라 변수 값만 바뀌는 방식임

왜 이렇게 사용하는지

밝은 테마와 어두운 테마를 따로 전부 만들 필요가 없음

같은 클래스 구조를 유지하면서 색상만 바꿀 수 있어서 코드가 짧아짐

전체 배치 CSS

왼쪽 메뉴와 오른쪽 내용을 grid로 나누는 부분임

실제 코드

```
.app-shell {
  display: grid;
  grid-template-columns: 260px minmax(0, 1fr);
  min-height: 100vh;
}

.sidebar {
  position: sticky;
  top: 0;
  height: 100vh;
}
```

어떻게 쓰는 코드인지

grid-template-columns는 왼쪽을 260px로 고정하고 오른쪽을 남은 공간 전체로 쓰게 함

sticky는 스크롤을 해도 사이드바가 화면에 붙어 있게 만듦

왜 이렇게 사용하는지

카페 사이트처럼 메뉴가 항상 보이면 사용자가 이동하기 편함

왼쪽 고정 메뉴와 오른쪽 콘텐츠 구조가 실제 커뮤니티 서비스 느낌을 줌

카드형 디자인

게시글과 패널을 보기 좋게 만드는 스타일임

실제 코드

```
.panel,
.post-card,
.modal-card {
  background: var(--surface);
  border: 1px solid var(--line);
  border-radius: 8px;
  box-shadow: var(--shadow);
}

.post-stack {
  display: grid;
  gap: 12px;
}
```

어떻게 쓰는 코드인지

panel과 post-card는 흰색 박스, 테두리, 그림자를 같이 사용함

post-stack은 게시글 카드 사이 간격을 자동으로 맞춤

왜 이렇게 사용하는지

같은 카드 스타일을 여러 기능에 재사용하면 화면이 통일됨

게시글, 관리자 목록, 최근 이야기 카드가 같은 서비스 안에 있는 것처럼 보이게 됨

버튼 스타일

primary, ghost, chip으로 버튼 종류를 나눈 구조임

실제 코드

```
.primary-btn {
  background: var(--brand);
  color: white;
  font-weight: 750;
}

.ghost-btn,
.chip {
  background: var(--surface);
  color: var(--ink);
  border-color: var(--line);
}

.chip.active {
  background: var(--brand-dark);
  color: white;
}
```

어떻게 쓰는 코드인지

primary-btn은 저장, 글쓰기처럼 중요한 버튼에 사용함

ghost-btn은 로그인, 회원가입처럼 보조 버튼에 사용함

chip은 게시판 필터처럼 선택 버튼에 사용함

왜 이렇게 사용하는지

버튼 역할에 따라 색을 나눠두면 사용자가 무엇을 먼저 눌러야 하는지 바로 알 수 있음

기능이 많아져도 버튼 스타일이 섞이지 않음

JavaScript 기본 데이터

seedState로 처음 실행될 때 필요한 데이터를 넣어둔 부분임

실제 코드

```
const STORE_KEY = "smc-cafe-state-v1";
const SESSION_KEY = "smc-cafe-session";

const seedState = {
  settings: { name: "SMC Cafe", intro: "함께 쓰는 인터넷 카페" },
  users: [
    { id: "smc", nickname: "SMC 관리자", role: "admin" },
    { id: "miso", nickname: "미소", role: "member" }
  ],
  posts: [],
  reports: [],
  chatRooms: []
};
```

어떻게 쓰는 코드인지

STORE_KEY는 전체 카페 데이터를 저장할 localStorage 이름임

SESSION_KEY는 현재 로그인한 사용자를 기억하는 이름임

seedState는 처음 접속했을 때 기본 회원, 게시글, 대화방을 만드는 초기 데이터임

왜 이렇게 사용하는지

처음 실행했을 때 화면이 비어 있으면 테스트하기 어려움

기본 데이터를 넣어두면 게시판, 멤버, 대화방 기능이 바로 보이기 때문에 개발과 발표에 유리함

localStorage 불러오기

브라우저 안에 저장된 데이터를 다시 읽는 함수임

실제 코드

```
function loadState() {
  const raw = localStorage.getItem(STORE_KEY);
  if (!raw) {
    const fresh =
normalizeState(structuredClone(seedState));
    localStorage.setItem(STORE_KEY,
JSON.stringify(fresh));
    return fresh;
  }
  const parsed = normalizeState(JSON.parse(raw));
  localStorage.setItem(STORE_KEY,
JSON.stringify(parsed));
  return parsed;
}
```

어떻게 쓰는 코드인지

localStorage에 데이터가 없으면 seedState를 복사해서 새로 저장함

이미 데이터가 있으면 JSON 문자열을 객체로 바꿔서 사용함

normalizeState를 거쳐서 누락된 값도 보정함

왜 이렇게 사용하는지

새로고침을 해도 글과 회원 정보가 사라지지 않게 하기 위한 구조임

서버 데이터베이스가 없어도 브라우저 안에서 간단한 저장 기능을 만들 수 있음

데이터 저장 함수

state가 바뀌면 localStorage에 다시 넣는 함수임

실제 코드

```
function saveState() {
  normalizeState(state);
  try {
    localStorage.setItem(STORE_KEY,
JSON.stringify(state));
    return true;
  } catch (error) {
    toast("저장 공간이 부족합니다. 이미지 크기나 개수를 줄여
주세요.");
    return false;
  }
}
```

어떻게 쓰는 코드인지

state는 현재 사이트의 전체 데이터임

JSON.stringify로 객체를 문자열로 바꾼 뒤 localStorage에 저장함

이미지가 너무 크면 저장 실패가 날 수 있어서 catch로 안내함

왜 이렇게 사용하는지

브라우저 저장소는 문자열만 저장할 수 있기 때문에 JSON 문자열 변환이 필요함

try catch를 넣어두면 오류가 생겼을 때 사이트가 갑자기 멈추는 것을 줄일 수 있음

데이터 보정 함수

오래된 데이터에도 새 기능이 붙도록 기본값을 채우는 부분임

실제 코드

```
function normalizeState(next) {
  next.settings ||= { name: "SMC Cafe", intro: "함께 쓰는 인터넷 카페" };
  next.users ||= [];
  next.posts ||= [];
  next.reports ||= [];
  next.chatRooms ||= [];

  next.posts.forEach((post) => {
    post.likes ||= [];
    post.dislikes ||= [];
    post.comments ||= [];
  });
  return next;
}
```

어떻게 쓰는 코드인지

예전 데이터에 likes나 comments가 없으면 빈 배열을 넣어줌

새 기능을 추가해도 기존 저장 데이터가 깨지지 않게 만드는 안전장치임

왜 이렇게 사용하는지

프로젝트를 계속 수정하다 보면 데이터 구조가 바뀜

normalizeState가 없으면 예전 데이터에서 undefined 오류가 생길 수 있어서 이런 보정 함수가 필요함

현재 사용자 확인

로그인한 사람과 관리자 여부를 판단하는 함수임

실제 코드

```
function currentUser() {
  return state.users.find((user) => user.id ===
currentUserId) || null;
}

function isAdmin() {
  return currentUser()?.role === "admin";
}

function userName(id) {
  return state.users.find((user) => user.id ===
id)?.nickname || "알 수 없음";
}
```

어떻게 쓰는 코드인지

currentUser는 현재 로그인한 사용자 객체를 찾아줌

isAdmin은 role 값이 admin인지 확인함

userName은 게시글 작성자 아이디를 닉네임으로 바꿔 보여줌

왜 이렇게 사용하는지

권한 처리를 여러 곳에서 직접 쓰면 코드가 길어짐

함수로 묶어두면 글 삭제, 관리자 메뉴, 공지 작성 같은 곳에서 같은 기준을 재사용할 수 있음

화면 이동 함수

버튼을 누르면 어떤 view를 보여줄지 바꾸는 부분임

실제 코드

```
function routeTo(route) {
  if (route === "admin" && !isAdmin()) {
    return toast("관리자만 들어갈 수 있습니다.");
  }
  if (route === "profile" && !currentUser()) {
    return openAuth("login");
  }
  currentRoute = route;
  $$(".nav-item").forEach((item) =>
item.classList.toggle("active", item.dataset.route ===
route));
  render();
}
```

어떻게 쓰는 코드인지

route 값이 admin이면 먼저 관리자 권한을 확인함

profile은 로그인해야 볼 수 있기 때문에 로그인 모달을 열게 함

현재 route를 바꾼 뒤 render로 화면을 다시 그림

왜 이렇게 사용하는지

여러 html 파일로 이동하지 않고도 화면 전환이 가능함

권한 검사와 화면 이동을 한 함수에 모아두면 관리하기 쉬운 구조가 됨

전체 렌더링 구조

데이터가 바뀌면 화면 전체를 다시 그리는 중심 함수임

실제 코드

```
function render() {
  renderHeader();
  renderHome();
  renderBoard();
  renderNotices();
  renderMembers();
  renderChat();
  renderProfile();
  if (isAdmin()) renderAdmin();

  $$(".view").forEach((view) =>
view.classList.remove("active-view"));
  $("#" + currentRoute + "View")?.classList.add("active-
view");
}
```

어떻게 쓰는 코드인지

render는 홈, 게시판, 공지, 멤버, 채팅, 프로필을 한 번에 갱신함

마지막에 currentRoute에 맞는 view만 active-view로 보여줌

왜 이렇게 사용하는지

글을 쓰거나 좋아요를 누르거나 로그인 상태가 바뀌면 화면도 같이 바뀌어야 함

render 하나로 갱신 흐름을 모으면 기능별 코드가 복잡하게 흩어지지 않음

게시글 카드 만들기

게시글 하나를 카드 HTML로 바꾸는 함수임

실제 코드

```
function renderPostCard(post) {
  const commentCount = post.comments.length;
  const canManage = isAdmin() || post.authorId ===
currentUserId;

  let html = "<article class='post-card'>";
  html += "<h3>" + escapeHtml(post.title) + "</h3>";
  html += "<p>" + escapeHtml(post.body) + "</p>";
  html += "<span>좋아요 " + post.likes.length + "
</span>";
  html += "<span>댓글 " + commentCount + "</span>";
  if (canManage) html += "<button>삭제</button>";
  html += "</article>";
  return html;
}
```

어떻게 쓰는 코드인지

게시글 데이터 하나를 받아서 화면에 들어갈 HTML 문자열을 만듦

canManage가 true일 때만 수정과 삭제 버튼을 넣음

escapeHtml을 써서 사용자가 입력한 글을 안전하게 출력함

왜 이렇게 사용하는지

게시글 목록과 최근 이야기에서 같은 카드 모양을 재사용할 수 있음

권한에 따라 버튼이 보이고 안 보이는 것도 카드 생성 단계에서 처리 가능함

검색과 카테고리 필터

게시글을 조건에 맞게 골라내는 함수임

실제 코드

```
function filteredPosts() {
  const keyword =
$("#globalSearch").value.trim().toLowerCase();
  return state.posts
    .filter((post) => currentCategory === "all" ||
post.category === currentCategory)
    .filter((post) => {
      if (!keyword) return true;
      const haystack = [post.title, post.body,
userName(post.authorId), post.tags.join(" ")]
        .join(" ")
        .toLowerCase();
      return haystack.includes(keyword);
    })
    .sort((a, b) => Number(b.pinned) -
Number(a.pinned) || b.createdAt - a.createdAt);
}
```

어떻게 쓰는 코드인지

현재 선택한 카테고리와 검색어를 기준으로 글을 걸러냄

haystack은 제목, 내용, 작성자, 태그를 한 문자열로 합친 검색 대상임

공지 고정 글이 위로 오고 그 다음 최신순으로 정렬됨

왜 이렇게 사용하는지

검색 대상을 한 번에 합치면 여러 필드를 따로 비교하지 않아도 됨

정렬 기준을 함수 안에 넣으면 게시판 화면이 항상 같은 규칙으로 보임

상세 보기 함수

글을 눌렀을 때 모달에 자세한 내용을 보여주는 부분임

실제 코드

```
function openPost(id, countView = true) {
  const post = state.posts.find((item) => item.id === id);
  if (!post) return;

  if (countView && !isPostRead(post)) {
    post.views += 1;
    post.readBy.push(currentUserId || "guest");
    saveState();
  }

  $("#postDetail").innerHTML =
    "<h2>" + escapeHtml(post.title) + "</h2>" +
    "<p>" + escapeHtml(post.body) + "</p>";
  openModal("detailModal");
}
```

어떻게 쓰는 코드인지

post id로 해당 게시글을 찾은 뒤 상세 모달에 내용을 넣음

처음 읽는 사용자라면 views를 증가시키고 readBy에 기록함

마지막에 detailModal을 열어서 보여줌

왜 이렇게 사용하는지

목록 화면에서는 제목과 일부 정보만 보여주고, 상세 화면에서 본문과 댓글을 보여주는 방식임

조회수 중복 증가를 줄이기 위해 readBy 배열을 같이 쓰는 구조임

글쓰기 모달 열기

새 글과 수정 글을 같은 폼으로 처리하는 방식임

실제 코드

```
function openPostForm(post = null, category = null) {
  if (!requireLogin()) return;
  $("#postModalTitle").textContent = post ? "글 수정" :
"새 글";
  $("#editingPostId").value = post?.id || "";
  $("#postCategory").value = category ||
post?.category || "free";
  $("#postTitle").value = post?.title || "";
  $("#postBody").value = post?.body || "";
  $("#postYoutube").value = post?.youtubeUrl || "";
  $("#postTags").value = post?.tags.join(", ") || "";
  openModal("postModal");
}
```

어떻게 쓰는 코드인지

post가 있으면 수정 모드이고 없으면 새 글 모드임

editingPostId에 값이 들어가면 저장할 때 기존 글 수정으로 처리됨

기존 제목과 내용을 폼에 미리 채워 넣는 구조임

왜 이렇게 사용하는지

새 글 작성과 글 수정을 다른 화면으로 만들 필요가 없음

같은 폼을 재사용하면 코드도 줄고 사용 방식도 통일됨

글 저장 처리

저장 버튼을 눌렀을 때 새 글을 만들거나 기존 글을 수정하는 부분임

실제 코드

```
$("#postForm").addEventListener("submit", async
(event) => {
  event.preventDefault();
  if (!requireLogin()) return;
  const category = $("#postCategory").value;
  if (category === "notice" && !isAdmin()) return
toast("공지는 관리자만 작성할 수 있습니다.");

  const payload = {
    category,
    title: $("#postTitle").value.trim(),
    body: $("#postBody").value.trim(),
    tags: $("#postTags").value.split(",").map((tag) =>
tag.trim()).filter(Boolean)
  };
});
```

어떻게 쓰는 코드인지

submit 이벤트는 저장 버튼을 눌렀을 때 실행됨

event.preventDefault로 폼이 새로고침되는 기본 동작을 막음

공지 카테고리는 관리자만 작성 가능하도록 한 번 더 검사함

왜 이렇게 사용하는지

글 저장은 권한 검사가 꼭 필요함

화면에서 버튼을 숨겨도 개발자 도구로 조작할 수 있기 때문에 저장 단계에서도 검사하는 방식이 더 안전함

새 글 추가 로직

state.posts 배열 앞에 새 게시글을 넣는 부분임

실제 코드

```
state.posts.unshift({
  id: uid(),
  ...payload,
  authorId: currentUserId,
  createdAt: Date.now(),
  likes: [],
  dislikes: [],
  scraps: [],
  readBy: [currentUserId],
  views: 0,
  pinned: category === "notice",
  comments: []
});

saveState();
render();
```

어떻게 쓰는 코드인지

unshift는 배열의 맨 앞에 새 값을 넣음

그래서 새 글이 게시판 가장 위에 바로 보임

좋아요, 싫어요, 댓글 같은 값은 새 글일 때 빈 배열로 시작함

왜 이렇게 사용하는지

커뮤니티 게시판은 보통 최신 글이 위에 올라옴

unshift를 쓰면 따로 정렬하지 않아도 새로 쓴 글이 앞쪽에 배치되는 효과가 있음

이미지 업로드 처리

이미지 파일을 브라우저에서 바로 보여주기 위해 Data URL로 바꾸는 부분임

실제 코드

```
function imageToDataUrl(file, maxWidth = 1280,
quality = 0.78) {
  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = () => {
      const img = new Image();
      img.onload = () => {
        const canvas =
document.createElement("canvas");
        // 이미지 크기 조절 뒤 base64로 저장함
      };
      img.src = reader.result;
    };
    reader.readAsDataURL(file);
  });
}
```

어떻게 쓰는 코드인지

FileReader는 사용자가 선택한 이미지 파일을 브라우저가 읽게 해줌

canvas는 이미지 크기를 줄이거나 압축해서 저장 용량을 줄이는 데 쓰임

결과는 base64 문자열 형태로 저장됨

왜 이렇게 사용하는지

localStorage는 파일 자체를 저장하지 못함

그래서 이미지를 문자열처럼 저장할 수 있는 Data URL 형태로 바꾸는 방식이 사용됨

유튜브 링크 처리

일반 유튜브 주소에서 영상 id만 뽑는 함수임

실제 코드

```
function extractYoutubeId(url = "") {
  const patterns = [
    /youtube.com/watch?v=([^&]+)/,
    /youtu.be/([^?]+)/,
    /youtube.com/embed/([^?]+)/
  ];
  for (const pattern of patterns) {
    const match = url.match(pattern);
    if (match) return match[1];
  }
  return "";
}

function youtubeEmbedUrl(url = "") {
  const id = extractYoutubeId(url);
  return id ? "https://www.youtube.com/embed/" + id :
"";
}
```

어떻게 쓰는 코드인지

watch 주소, youtu.be 주소, embed 주소를 모두 처리하려고 정규식을 여러 개 둠

영상 id를 찾으면 iframe에 넣을 embed 주소로 바꿈

왜 이렇게 사용하는지

사용자가 입력하는 유튜브 링크 형태는 하나로 고정되지 않음

여러 주소 형태를 받아주면 사용자가 링크를 잘못 넣는 경우를 줄일 수 있음

좋아요와 싫어요

같은 사용자가 중복으로 누르지 않게 배열을 토글하는 구조임

실제 코드

```
function toggleListValue(list, value) {
  return list.includes(value)
    ? list.filter((item) => item !== value)
    : [...list, value];
}

if (target.dataset.like && post) {
  post.likes = toggleListValue(post.likes,
currentUserId);
  if (post.likes.includes(currentUserId)) {
    post.dislikes = post.dislikes.filter((id) => id !==
currentUserId);
  }
  saveState();
  render();
}
```

어떻게 쓰는 코드인지

likes 배열에는 좋아요를 누른 사용자 id가 들어감

이미 누른 사람이 다시 누르면 배열에서 제거됨

좋아요를 누르면 싫어요 배열에서는 같은 사용자를 빼서 동시에 둘 다 못 누르게 함

왜 이렇게 사용하는지

숫자만 올리는 방식이면 같은 사람이 계속 누를 수 있음

사용자 id 배열로 관리하면 누가 눌렀는지 확인할 수 있어서 중복 방지가 가능함

댓글 등록

상세 보기 아래 입력창에서 댓글을 저장하는 부분임

실제 코드

```
document.addEventListener("submit", (event) => {
  const form = event.target.closest("[data-comment-form]");
  if (!form) return;
  event.preventDefault();
  if (!requireLogin()) return;

  const input = form.querySelector("input");
  const body = input.value.trim();
  if (!body) return;

  const post = state.posts.find((item) => item.id === form.dataset.commentForm);
  post.comments.push({ id: uid(), authorId: currentUserId, body, createdAt: Date.now() });
  saveState();
  openPost(post.id, false);
});
```

어떻게 쓰는 코드인지

data-comment-form에는 어떤 게시글에 댓글을 달지 post id가 들어 있음

댓글 내용이 비어 있으면 저장하지 않음

저장 후 openPost를 다시 호출해서 상세 화면을 갱신함

왜 이렇게 사용하는지

댓글은 게시글 안에 포함된 하위 데이터라서 post.comments 배열에 넣음

상세 모달을 다시 그리면 새 댓글이 바로 보이는 구조가 됨

신고 기능

사용자가 글을 신고하면 reports 배열에 기록하는 구조임

실제 코드

```
if (target.dataset.report) {
  if (!requireLogin()) return;
  const reason = prompt("신고 사유를 입력해주세요.");
  if (!reason) return;

  state.reports.unshift({
    id: uid(),
    postId: target.dataset.report,
    reporterId: currentUserId,
    reason,
    status: "open",
    createdAt: Date.now()
  });
  saveState();
  render();
}
```

어떻게 쓰는 코드인지

신고 버튼을 누르면 사유를 입력받음

postId는 어떤 글인지, reporterId는 누가 신고했는지 기록함

status는 처음에는 open으로 저장됨

왜 이렇게 사용하는지

신고는 바로 글을 삭제하는 기능이 아니라 관리자에게 검토할 자료를 남기는 기능임

관리자 화면에서 status를 바꾸며 처리하는 흐름으로 만들 수 있음

로그인 처리

아이디와 비밀번호가 맞는 사용자를 찾아 세션을 저장하는 부분임

실제 코드

```
if (authMode === "login") {
  const user = state.users.find((item) => item.id === id
&& item.password === password);
  if (!user) return toast("아이디나 비밀번호를 확인해주세
요.");
  if (!user.approved) return toast("관리자 승인 후 로그인
할 수 있습니다.");
  if (user.blocked) return toast("이용 제한된 계정입니다.");

  currentUserId = user.id;
  localStorage.setItem(SESSION_KEY, currentUserId);
  closeModal("authModal");
  toast(user.nickname + "님, 반갑습니다.");
}
```

어떻게 쓰는 코드인지

입력한 id와 password가 둘 다 일치하는 사용자를 찾음

승인되지 않은 계정과 차단된 계정은 로그인하지 못하게 막음

로그인 성공 시 currentUserId와 SESSION_KEY에 사용자 id를 저장함

왜 이렇게 사용하는지

로그인 상태를 localStorage에 저장하면 새로고침 후에도 사용자를 기억할 수 있음

관리자 승인과 차단 검사를 넣어서 회원 관리 기능과 연결됨

회원가입 처리

새 사용자를 users 배열에 추가하는 부분임

실제 코드

```
if (state.users.some((user) => user.id.toLowerCase()
=== id.toLowerCase())) {
  return toast("이미 있는 아이디입니다.");
}
if (password !== passwordConfirm) {
  return toast("비밀번호 확인이 맞지 않습니다.");
}

state.users.push({
  id,
  password,
  nickname,
  role: "member",
  joinedAt: new Date().toISOString().slice(0, 10),
  approved: false,
  blocked: false,
  memo: "가입 승인 대기"
});
```

어떻게 쓰는 코드인지

이미 있는 아이디인지 먼저 확인함

비밀번호와 비밀번호 확인이 같아야 가입됨

새 회원은 approved가 false라서 관리자 승인 전에는 로그인할 수 없음

왜 이렇게 사용하는지

커뮤니티는 아무나 바로 들어오게 할 수도 있지만 이 프로젝트는 관리자 승인 구조를 넣었음

회원 관리 기능을 보여주기 좋은 방식임

관리자 화면 렌더링

신고, 회원, 게시글, 문의를 관리자 화면에 표시하는 부분임

실제 코드

```
function renderAdmin() {
  $("#reportList").innerHTML = state.reports
    .map((report) => {
      const post = state.posts.find((item) => item.id === report.postId);
      return "<div class='admin-row'>"
        + "<strong>" + escapeHtml(post?.title || "삭제된 글") + "</strong>"
        + "<p>" + escapeHtml(report.reason) + "</p>"
        + "<button>처리 완료</button>"
        + "</div>";
    }).join("");
}
```

어떻게 쓰는 코드인지

state.reports 안의 신고 목록을 관리자 카드로 바꿈

신고된 글 제목과 신고 사유를 보여줌

처리 완료 버튼으로 신고 상태를 바꿀 수 있음

왜 이렇게 사용하는지

관리자는 일반 회원과 달리 사이트 전체 데이터를 봐야 함

신고, 회원, 게시글을 한 화면에 모아두면 실제 카페 관리 페이지처럼 작동함

대화방 구조

대화방과 메시지를 chatRooms 배열 안에 저장하는 방식임

실제 코드

```
chatRooms: [
  {
    id: "main",
    name: "전체 대화방",
    ownerId: "smc",
    messages: [
      { id: uid(), type: "notice", authorId: "smc", body: "오늘도 편하게 이용해주세요." },
      { id: uid(), type: "message", authorId: "miso", body: "새 게시판 보기 좋아졌네요." }
    ]
  }
]
```

어떻게 쓰는 코드인지

chatRooms는 방 목록이고, 각 방 안에 messages 배열이 들어감

메시지는 일반 message, notice, poll 같은 type으로 나뉨

방을 여러 개 만들 수 있게 되어 있음

왜 이렇게 사용하는지

대화방마다 메시지를 따로 저장해야 방을 바꿨을 때 내용이 섞이지 않음

messages를 방 내부에 넣어두면 구조가 직관적임

투표 메시지

채팅 안에서 투표를 만들고 참여 수를 계산하는 부분임

실제 코드

```
if (chat.type === "poll") {
  const total = Object.keys(chat.votes || {}).length;
  let html = "<div class='chat-bubble poll-bubble'>";
  html += "<strong>투표</strong>";
  html += "<p>" + escapeHtml(chat.body) + "</p>";
  chat.options.forEach((option) => {
    const count = Object.values(chat.votes ||
{}).filter((vote) => vote === option).length;
    html += "<button>" + option + " " + count + "
</button>";
  });
  html += "<span>참여 " + total + "명</span>";
  html += "</div>";
  return html;
}
```

어떻게 쓰는 코드인지

투표 메시지는 options와 votes 데이터를 사용함

votes는 사용자 id별로 선택한 항목을 저장함

각 option마다 몇 명이 골랐는지 계산해서 화면에 표시함

왜 이렇게 사용하는지

채팅은 단순 메시지만 있으면 기능이 평범함

투표 기능을 넣으면 대화방에서 의견을 모으는 커뮤니티 기능으로 확장됨

프로필 화면

내가 쓴 글, 댓글, 좋아요, 스크랩을 모아 보여주는 부분임

실제 코드

```
function renderProfile() {
  const user = currentUser();
  if (!user) {
    $("#profilePanel").innerHTML = "로그인 후 내 프로필을
볼 수 있습니다.";
    return;
  }

  const myPosts = state.posts.filter((post) =>
post.authorId === user.id);
  const myComments = state.posts.flatMap((post) =>
post.comments.filter((comment) =>
comment.authorId === user.id));
  const likedPosts = state.posts.filter((post) =>
post.likes.includes(user.id));
}
```

어떻게 쓰는 코드인지

현재 로그인한 사용자를 기준으로 내가 쓴 글과 댓글을 계산함

좋아요 누른 글과 스크랩한 글도 따로 모음

이 결과를 프로필 화면 카드에 표시함

왜 이렇게 사용하는지

프로필은 단순 개인정보 화면이 아니라 사용자의 활동 기록을 보여주는 공간임

커뮤니티에서 내가 남긴 흔적을 모아 볼 수 있게 하는 기능임

카페 설정 저장

관리자가 카페 이름과 소개 문구를 바꾸는 부분임

실제 코드

```
$("#settingsForm").addEventListener("submit", async (event) => {
  event.preventDefault();
  if (!isAdmin()) return;

  state.settings.name = $("#settingName").value.trim() || "SMC Cafe";
  state.settings.intro = $("#settingIntro").value.trim() || "함께 쓰는 인터넷 카페";
  saveState();
  render();
  toast("카페 설정을 저장했습니다.");
});
```

어떻게 쓰는 코드인지

관리자 화면의 설정 폼에서 카페 이름과 소개 문구를 읽음

값이 비어 있으면 기본값으로 저장함

저장 후 render를 호출해서 홈 화면 문구가 바로 바뀌게 함

왜 이렇게 사용하는지

카페 이름 같은 설정은 코드 안에 고정하면 매번 파일을 수정해야 함

settings 객체로 분리하면 관리자 화면에서 직접 바꿀 수 있는 구조가 됨

JSON 데이터 형식

서버 저장용 posts.json은 배열 형태로 준비되어 있음

실제 코드

```
[
  {
    "id": 1720000000000,
    "title": "게시글 제목",
    "content": "게시글 내용",
    "board": "자유게시판",
    "author": "익명",
    "image": "",
    "youtube": "",
    "views": 0,
    "likes": 0,
    "dislikes": 0
  }
]
```

어떻게 쓰는 코드인지

JSON은 JavaScript 객체와 비슷하게 생긴 데이터 저장 형식임

문자열은 큰따옴표로 감싸고, 배열과 객체 구조로 데이터를 정리함

posts.json은 게시글 목록을 저장하기 위해 준비된 파일임

왜 이렇게 사용하는지

서버에서 데이터를 파일로 저장하려면 사람이 읽기 쉽고 프로그램도 읽기 쉬운 형식이 필요함

JSON은 웹 개발에서 가장 많이 쓰이는 데이터 교환 형식이라 사용하기 좋음

Express 서버 시작

Node.js로 웹서버를 여는 server.js의 기본 구조임

실제 코드

```
const express = require("express");
const fs = require("fs");
const path = require("path");

const app = express();
const PORT = 3000;

const dataDir = path.join(__dirname, "data");
const postsFile = path.join(dataDir,
"posts.json");

app.use(express.json({ limit: "10mb" }));
app.use(express.static(path.join(__dirname,
"public")));
```

어떻게 쓰는 코드인지

express는 웹서버를 쉽게 만들기 위해 불러옴

fs는 posts.json 같은 파일을 읽고 쓰기 위해 사용함

path는 운영체제에 맞게 파일 경로를 안전하게 만들기 위해 사용함

왜 이렇게 사용하는지

프론트만 있으면 브라우저 안에서만 데이터가 저장됨

Express 서버를 붙이면 여러 사용자가 공유할 수 있는 API 구조로 확장할 수 있음

데이터 파일 자동 생성

posts.json이 없을 때 자동으로 만드는 부분임

실제 코드

```
if (!fs.existsSync(dataDir)) {
  fs.mkdirSync(dataDir);
}

if (!fs.existsSync(postsFile)) {
  fs.writeFileSync(postsFile, "[]", "utf-8");
}
```

어떻게 쓰는 코드인지

data 폴더가 없으면 mkdirSync로 새로 만듦

posts.json 파일이 없으면 빈 배열 문자열을 넣어서 생성함

서버 실행 전에 필요한 저장 공간을 준비하는 코드임

왜 이렇게 사용하는지

처음 프로젝트를 받은 사람은 posts.json 파일이 없을 수도 있음

자동 생성 코드를 넣으면 실행할 때 파일이 없어서 서버가 멈추는 문제를 줄일 수 있음

게시글 목록 API

브라우저가 서버에서 게시글을 받아올 수 있게 하는 주소임

실제 코드

```
app.get("/api/posts", (req, res) => {
  const posts =
JSON.parse(fs.readFileSync(postsFile, "utf-
8"));
  res.json(posts);
});
```

어떻게 쓰는 코드인지

GET /api/posts 주소로 요청이 오면 posts.json을 읽음

JSON.parse로 문자열을 배열로 바꾼 뒤 res.json으로 응답함

브라우저는 이 응답을 받아 게시판 목록을 만들 수 있음

왜 이렇게 사용하는지

API를 만들면 화면과 데이터 저장을 분리할 수 있음

HTML은 화면만 담당하고 서버는 데이터를 주고받는 역할을 하게 됨

게시글 추가 API

새 글을 서버 파일에 저장하는 주소임

실제 코드

```
app.post("/api/posts", (req, res) => {
  const { title, content, board, author, image, youtube }
= req.body;

  if (!title || !content) {
    return res.status(400).json({ message: "제목과 내용은
필수입니다." });
  }

  const posts = JSON.parse(fs.readFileSync(postsFile,
"utf-8"));
  const newPost = { id: Date.now(), title, content,
board, author, image, youtube };
  posts.unshift(newPost);
  fs.writeFileSync(postsFile, JSON.stringify(posts, null,
2), "utf-8");
});
```

어떻게 쓰는 코드인지

POST 요청으로 들어온 제목과 내용을 req.body에서 꺼냄

필수 값이 없으면 400 상태코드로 오류를 보냄

새 게시글을 posts 배열 앞에 넣고 파일에 다시 저장함

왜 이렇게 사용하는지

서버에 저장할 때도 입력값 검사가 필요함

프론트에서 막아도 요청을 직접 보낼 수 있기 때문에 서버 쪽에서 한 번 더 검사하는 구조가 맞음

서버 실행 코드

3000번 포트에서 사이트를 열어주는 마지막 부분임

실제 코드

```
app.listen(PORT, () => {
  console.log("서버 실행 중: http://localhost:" +
PORT);
});
```

어떻게 쓰는 코드인지

app.listen은 서버를 실제로 켜는 명령임

PORT가 3000이므로 브라우저에서 localhost:3000으로 접속함

console.log 문구로 서버가 정상 실행됐는지 터미널에서 확인할 수 있음

왜 이렇게 사용하는지

서버를 켜야 브라우저가 HTML, CSS, JS 파일을 받아올 수 있음

실행 문구가 있으면 초보자도 서버가 켜졌는지 바로 확인하기 쉬움

package.json

실행 명령과 의존성을 적어두는 프로젝트 설명 파일

실제 코드

```
{
  "name": "smc-cafe",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "start": "npx serve .",
    "preview": "npx serve .",
    "build": "echo Static site ready"
  },
  "main": "server.js",
  "dependencies": {
    "express": "^5.2.1"
  }
}
```

어떻게 쓰는 코드인지

scripts는 npm start 같은 명령을 정리하는 곳임

dependencies에는 프로젝트가 필요로 하는 라이브러리가 들어감

express가 적혀 있기 때문에 npm install 후 서버 코드를 실행할 수 있음

왜 이렇게 사용하는지

프로젝트를 다른 컴퓨터에서 열어도 같은 라이브러리를 설치할 수 있어야 함

package.json은 Node 프로젝트의 설명서이자 실행 설정 파일 역할을 함

Dockerfile

서버를 컨테이너로 실행하기 위한 설정임

실제 코드

```
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "server.js"]
```

어떻게 쓰는 코드인지

node:20-alpine 이미지를 기반으로 실행 환경을 만듦

/app 폴더를 작업 위치로 잡고 npm install을 실행함

마지막에 node server.js로 서버를 켬

왜 이렇게 사용하는지

내 컴퓨터에서는 되는데 다른 곳에서는 안 되는 문제를 줄이는 방식임

컨테이너에 실행 환경을 통째로 담으면 배포가 더 일정해짐

Netlify와 Vercel 설정

정적 사이트로 배포하기 위해 들어간 설정 파일임

실제 코드

netlify.toml

```
[build]
  publish = "."
  command = "echo Static site ready"
```

vercel.json

```
{
  "version": 2,
  "builds": [
    { "src": "**/*", "use": "@vercel/static" }
  ]
}
```

어떻게 쓰는 코드인지

netlify.toml은 Netlify 배포 설정임

vercel.json은 Vercel 배포 설정임

현재 구조는 정적 사이트처럼 올릴 수 있게 준비되어 있음

왜 이렇게 사용하는지

학교 발표나 포트폴리오에서는 실제 주소로 보여주는 것이 편함

배포 설정이 있으면 웹사이트를 빠르게 외부에 공개할 수 있음

직접 실행하면서 확인한 부분

코드만 읽은 게 아니라 실제 화면 흐름까지 같이 확인한 내용임

홈 화면	index.html의 homeView가 처음 active-view로 보여지는 것을 확인함
게시판	filter-row와 boardList가 renderBoard로 채워지는 흐름을 확인함
로그인	smc 관리자 계정으로 로그인했을 때 관리자 메뉴가 보이는 것을 확인함
글쓰기	제목과 내용을 넣고 저장하자 새 글이 맨 위에 추가되는 것을 확인함
관리자	신고함, 회원 관리, 카페 설정, 게시글 관리 영역이 렌더링되는 것을 확인함
대화방	메시지 종류를 일반, 공지, 투표로 나눠 보내는 UI를 확인함

코드상에서 더 손보면 좋은 부분

발표에서 질문받을 수 있는 개선 포인트도 같이 정리함

1. 저장 구조

현재 app.js 기능 대부분은 localStorage 중심임

server.js에도 API가 있지만 프론트 코드가 전부 서버 API와 연결된 구조는 아님

여러 사용자가 같은 데이터를 공유하려면 fetch로 API를 호출하게 바꾸는 작업이 필요함

2. 보안 구조

관리자 비밀번호가 코드 안에 들어가 있는 데모 구조임

실제 배포에서는 비밀번호를 코드에 직접 적지 않고 서버와 데이터베이스에서 안전하게 처리해야 함

비밀번호 해시 저장도 필요함

3. 파일 경로

index.html은 assets 이미지를 불러오지만 Express는 public 폴더만 정적으로 열고 있음

실행 환경에 따라 이미지 경로가 안 맞을 수 있음

assets를 public 안으로 옮기거나 server.js에서 assets 폴더도 열어주는 방식이 필요함

마무리

이 프로젝트는 단순한 화면만 있는 사이트가 아니라, 게시글 작성, 로그인, 댓글, 좋아요, 신고, 관리자, 대화방까지 들어간 커뮤니티 구조임

HTML은 화면의 뼈대, CSS는 디자인, JavaScript는 기능, JSON은 데이터 형식, Node.js와 Express는 서버 실행을 맡는 구조임

직접 실행해 보면 각 기능이 localStorage 상태값과 render 함수 중심으로 이어져 있다는 점이 핵심임

Q&A

감사합니다.