

# 배준형

UX / UI의 가치를 추구하는 개발자

---

📞 010-7123-0537

✉️ ryanbae94@gmail.com

🏡 ryanbae.dev

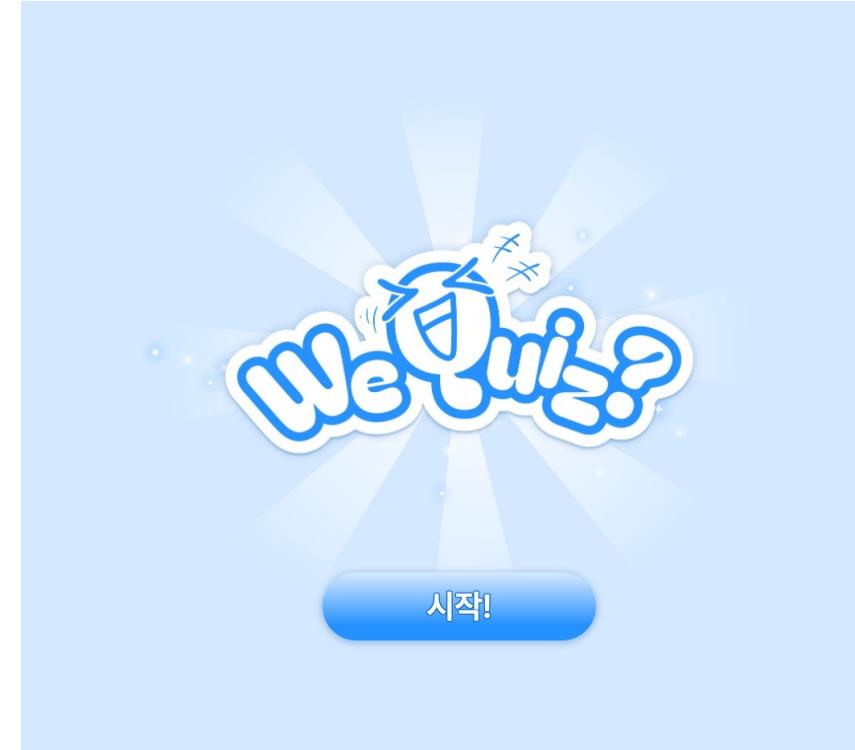
# CONTENTS

1. 프로젝트 소개
2. 전체 서비스 플로우
3. 메인 작업
4. 아키텍쳐
5. 회고

## 1. 프로젝트 소개

### 프로젝트 기본정보

프로젝트명	WeQuiz
프로젝트기간	2024.03~진행 중
프로젝트인원	5명 (FE 2 / BE 2 / ML 1) – FE 기여도 70%
설명	사용자가 업로드한 파일을 기반으로 ML을 통해 퀴즈를 생성하고, 소켓을 통해 실시간으로 퀴즈 게임을 함께 풀고 결과를 확인해볼 수 있는 웹 퀴즈 게임입니다.
담당역할	프론트엔드 개발
링크	배포: <a href="https://wequiz.kr">https://wequiz.kr</a> 깃허브: <a href="https://github.com/Team-WeQuiz/wequiz">https://github.com/Team-WeQuiz/wequiz</a>



[WeQuiz 시작 페이지]

### 개발 주요사항

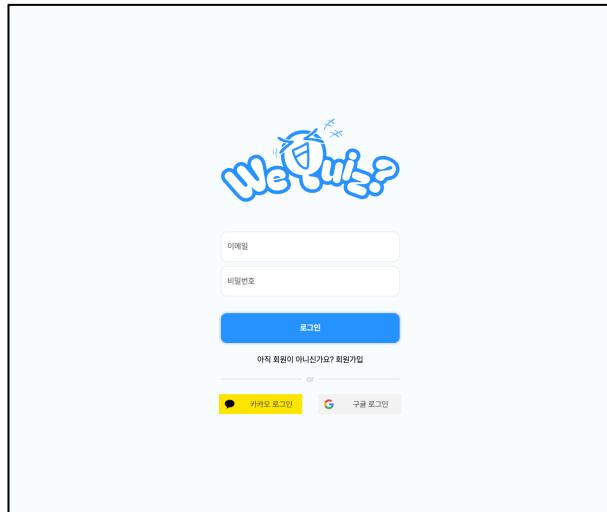
- 웹소켓을 활용한 실시간 소켓 통신 시스템
- 실시간 멀티플레이
- middleware를 활용한 라우팅 가드

## 1. 프로젝트 소개

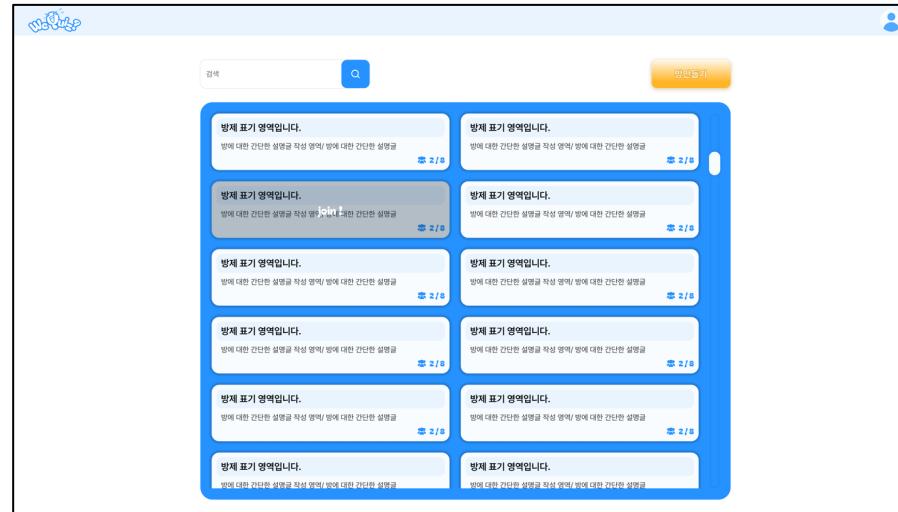
### 기술 스택 및 선정 이유

- Next.js 14 App router
  - ✓ SEO를 위한 SSR, 직관적인 app routing 시스템 사용을 위해 도입
- Vanilla-Extract
  - ✓ 런타임 오버헤드가 없는 CSS-IN-JS 활용을 위해 도입
- Zustand
  - ✓ 유저 로그인 정보 (AccessToken), bgm 세팅의 전역변수화를 위해 도입
  - ✓ 같은 FLUX 아키텍처를 사용하는 Redux에 비해 낮은 러닝 커브와 직관적인 사용법
- Axios
  - ✓ Fetch에 비해 간결한 사용법
  - ✓ JSON으로 파싱할 필요가 없어 코드 중복 감소
- Stomp.js
  - ✓ 실시간 통신을 위한 소켓 도입
- StoryBook
  - ✓ 컴포넌트 중심의 테스트 가능
  - ✓ 독립적인 컴포넌트 테스트 기능으로 생산성을 높일 수 있었음
  - ✓ UX와 관련해 팀원과 원활한 소통이 가능해짐
- TypeScript
  - ✓ 타입 안정성
  - ✓ 타입의 개별적인 정의 및 관리로, 받아오는 데이터의 재사용성을 높임
- Spring Boot
- MySQL
- DynamoDB
- MinIO
- Portainer
- AWS EC2
- AWS Secret Manager
- Fast API
- LangChain

## 2. 전체 서비스 플로우



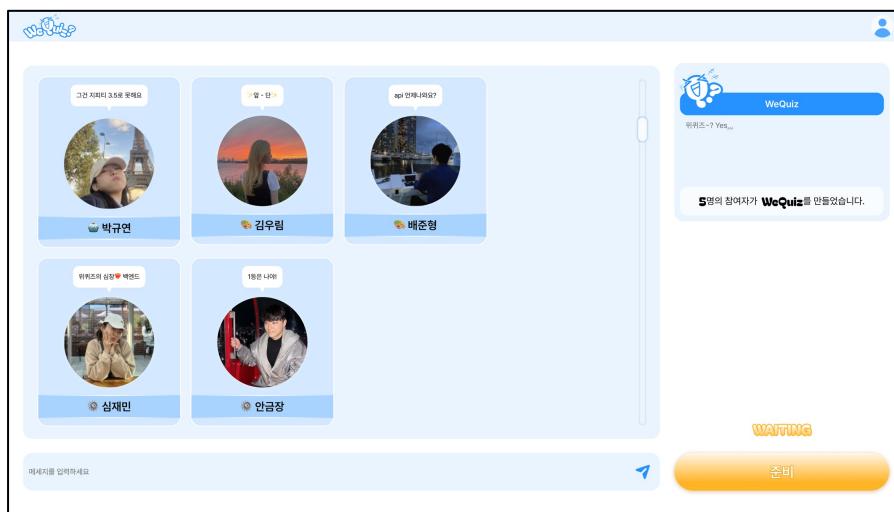
로그인



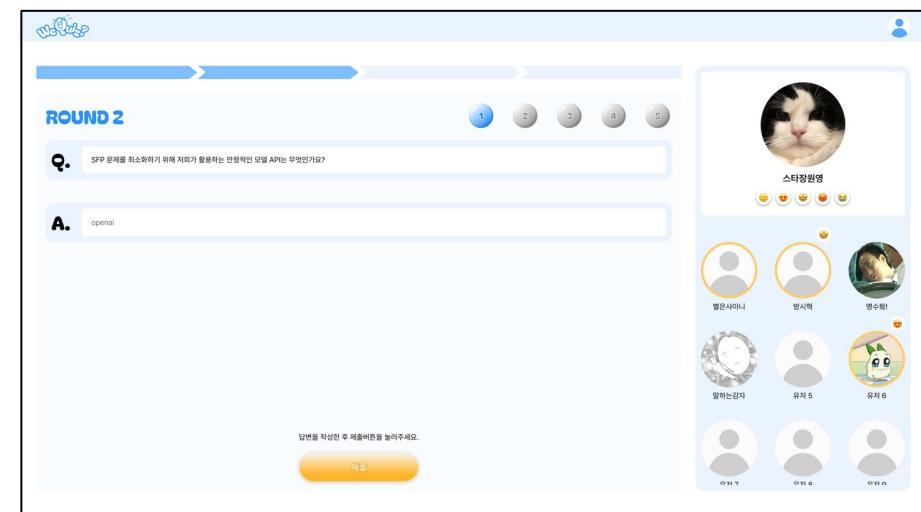
방 만들기 혹은 만들어진 방 입장 (초대 코드 입력 가능)



방 만들기

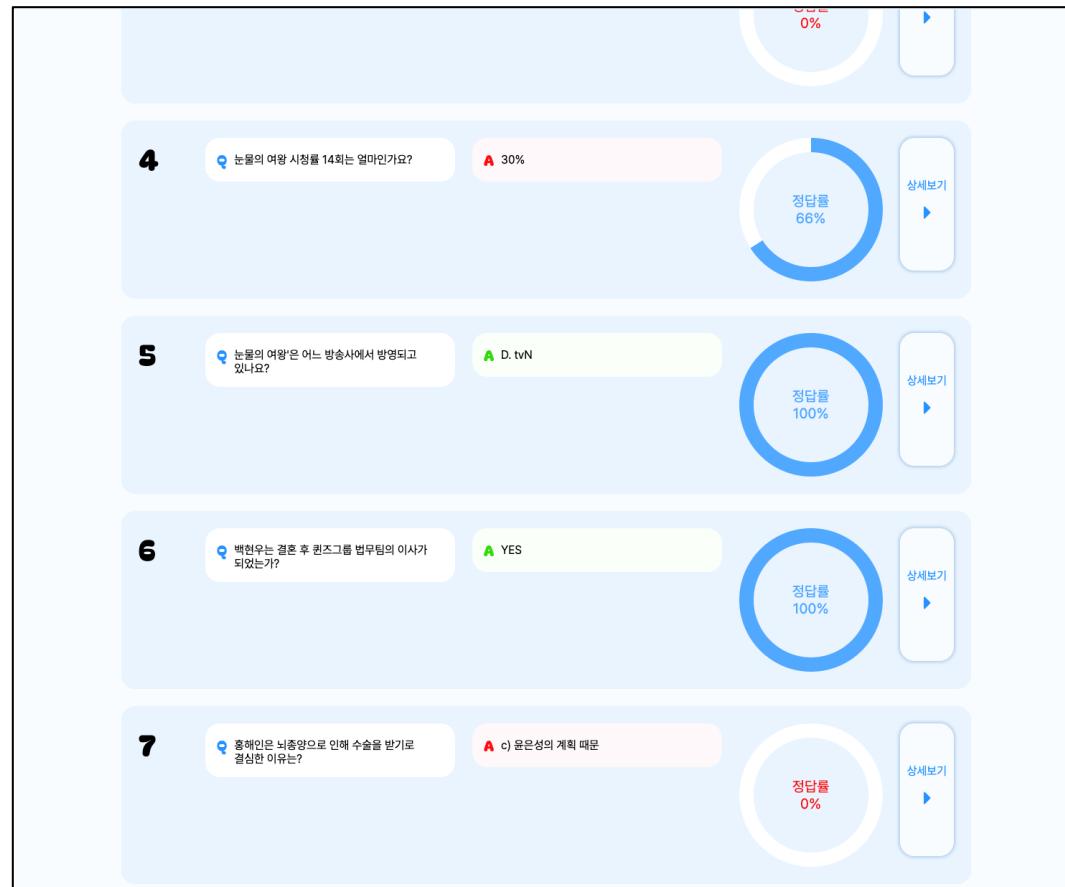


대기실 입장 (채팅 가능, 모든 유저 준비 후 시작)

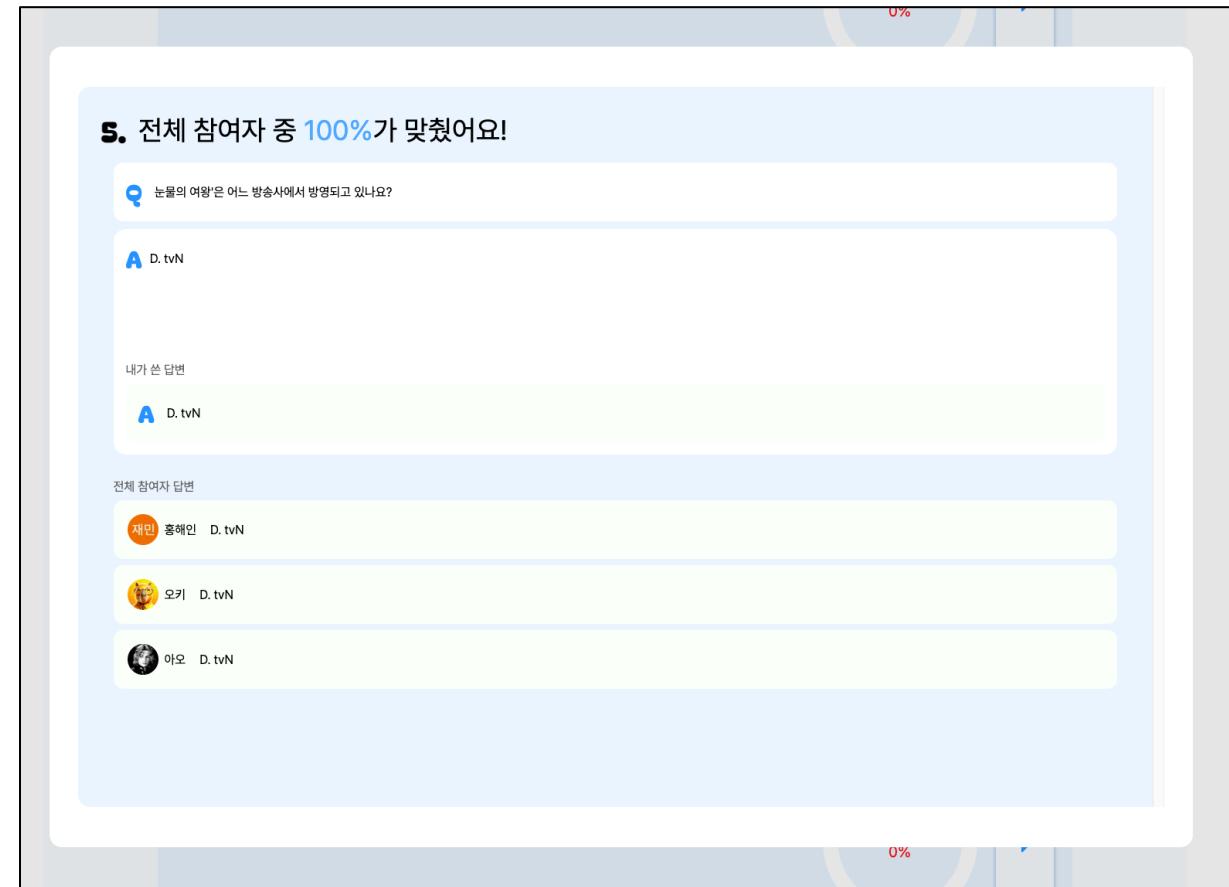


실시간 게임 진행 (이모티콘 소통 및 제출 상황 확인 가능)

## 2. 전체 서비스 플로우



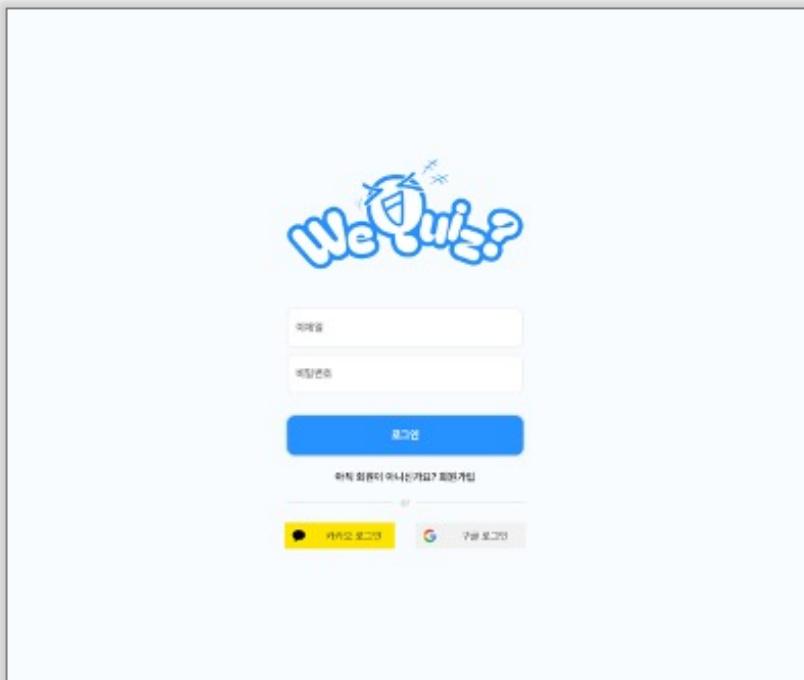
퀴즈 종료 후 결과 페이지



결과 페이지 상세 모달

### 3. 메인 작업

#### 로그인 / 회원가입 기능 (기여도 100%)



[로그인 / 회원가입 페이지]

##### 로그인 / 회원가입 퍼블리싱 및 기능 구현

- 이메일을 활용한 일반 로그인 및 회원가입
- kakao, google 로그인
- 활용되는 api key는 AWS Secret Manager를 활용하여 관리
- kakao sdk 로드 시 script 옵션을 lazyOnload -> beforeInteraction으로 변경하여 사용자가 sdk의 로드를 기다리지 않아도 되도록 개선하였습니다.

##### Auth 관리

- AccessToken은 클라이언트 메모리(전역변수)에 할당
- RefreshToken은 Cookie에 할당
- 새로고침 등의 이유로 AccessToken이 휘발되는 경우를 고려하여 cookie에 RefreshToken을 확인하고, 존재할 경우 다시 AccessToken을 재발급 받는 방식으로 구현하였습니다.

### 3. 메인 작업

#### 라우팅 가드 (기여도 100%)

```
const protectedPatterns = [
  '/main-lobby',
  '/create-room',
  '/waiting-room/:id'
].map(path => new URLPattern({ pathname: path }));

const publicRoutes = ['/sign-in', '/sign-up'];

export function middleware(request: NextRequest) {
  const token = request.cookies.get('refreshToken');
  const currentPath = request.nextUrl.pathname;

  const isProtectedRoute = protectedPatterns.some((pattern) =>
    pattern.test(request.nextUrl),
  );

  if (!token && isProtectedRoute) {
    const url = request.nextUrl.clone();
    url.pathname = '/sign-in';
    url.searchParams.set('message', '로그인이 필요한 페이지입니다.');
    return NextResponse.redirect(url);
  }

  if (token && publicRoutes.includes(currentPath)) {
    const url = request.nextUrl.clone();
    url.pathname = '/main-lobby';
    url.searchParams.set('message', '이미 로그인된 상태입니다.');
    return NextResponse.redirect(url);
  }

  const response = NextResponse.next();
  response.headers.set('Content-Security-Policy', 'upgrade-insecure-requests');

  return response;
}
```

[middleware.ts 코드]

#### middleware를 활용한 페이지 접근 제한

Next.js의 middleware 기능을 활용하여 서버사이드에서 로그인 여부 판단 후 페이지의 접근을 제한하였습니다.

클라이언트 사이드에서 로그인 여부 판단 시 발생하던 빈 캡데기 렌더링, 불필요한 api 호출의 문제를 해결하였습니다.

로그인이 필요한 페이지와 그렇지 않은 페이지를 각각 protectedPatterns, publicRoutes에 할당한 후, 요청된 pathname을 확인하여 cookie에 할당된 refreshToken의 존재 여부를 판단하는 방식으로 구현하였습니다.

### 3. 메인 작업

#### 방 만들기 페이지 (기여도 100%)



[방 만들기 페이지]

#### 파일 업로드

drag & drop 혹은 로컬 파일시스템의 파일을 선택할 수 있도록 구현하였습니다. 클라이언트 측에서 파일의 용량을 30mb로 제한하고, nginx에서 client\_max\_body\_size를 제한하여 ml 서버 자원이 낭비되는 일을 방지하였습니다.

### 3. 메인 작업

## S3 멀티파트 업로드 (기여도 100%)



```
import axios from 'axios';
import { createFileName } from './createFileName';

type CompletedPartType = {
  Etag: string;
  PartNumber: number;
};

// 파일을 파트로 쪼개고 해당 파일 생성
export const createChunkedArray = async (
  file: File,
  chunkSize: number,
): Promise<Blob[]> => {
  const chunkedArray: Blob[] = [];
  let offset = 0;

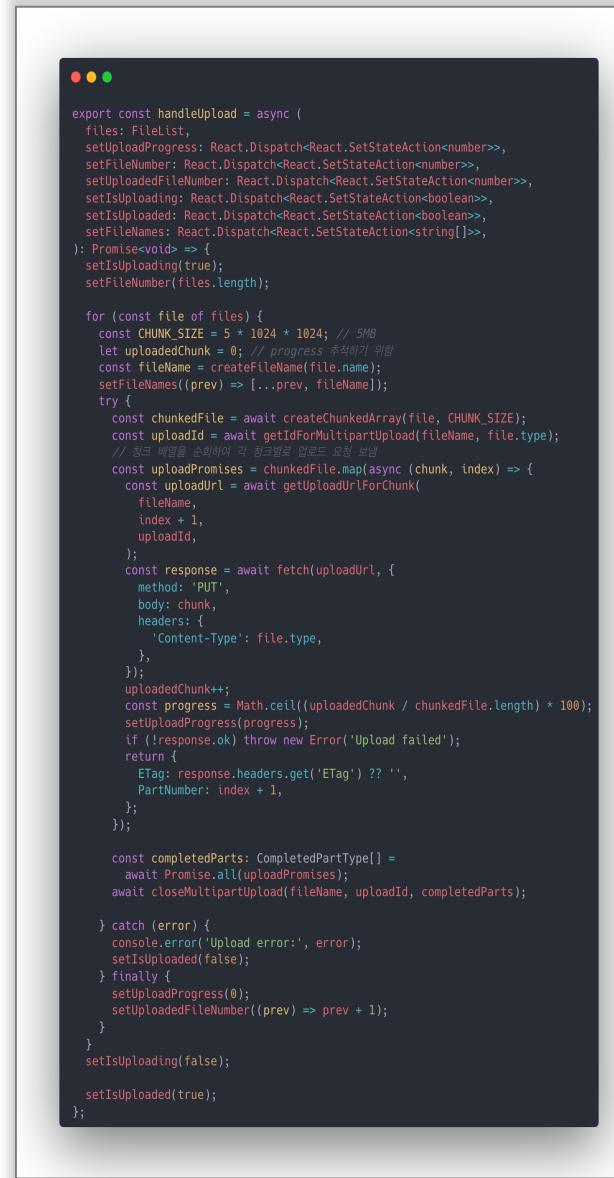
  while (offset < file.size) {
    const chunk = file.slice(offset, offset + chunkSize);
    chunkedArray.push(chunk);
    offset += chunkSize;
  }

  return chunkedArray;
};

// 멀티파트 업로드 할당
export const getIdForMultipartUpload = async (
  fileName: string,
  fileType: string,
): Promise<string> => {
  try {
    const response = await axios.post('/api/s3-upload/get-id', {
      data: { fileName, fileType },
      action: 'get-id',
    });
    return response.data.uploadId;
  } catch (error) {
    console.error('Error obtaining upload ID:', error);
    throw error;
  }
};

// 각 청크 별로 presigned url 요청
export const getUploadUrlForChunk = async (
  fileName: string,
  partNumber: number,
  uploadId: string,
): Promise<string> => {
  try {
    const response = await axios.post('/api/s3-upload/get-url', {
      data: {
        fileName,
        partNumber,
        uploadId,
      },
      action: 'get-url',
    });
    return response.data;
  } catch (error) {
    console.error('Error obtaining upload URL:', error);
    throw error;
  }
};

// 멀티파트 업로드 완료
export const closeMultipartUpload = async (
  fileName: string,
  uploadId: string,
  completedParts: CompletedPartType[],
): Promise<object | void> => {
  try {
    await axios.post('/api/s3-upload/complete', {
      data: { fileName, uploadId, parts: completedParts },
      action: 'complete-upload',
    });
  } catch (error) {
    console.error('Error completing upload:', error);
    throw error;
  }
};
```



```
export const handleUpload = async (
  files: FileList,
  setUploadProgress: React.Dispatch<React.SetStateAction<number>>,
  setFileNumber: React.Dispatch<React.SetStateAction<number>>,
  setUploadedFileNumber: React.Dispatch<React.SetStateAction<number>>,
  setIsUploading: React.Dispatch<React.SetStateAction<boolean>>,
  setIsUploaded: React.Dispatch<React.SetStateAction<boolean>>,
  setFileNames: React.Dispatch<React.SetStateAction<string[]>>,
): Promise<void> => {
  setIsUploading(true);
  setFileNumber(files.length);

  for (const file of files) {
    const CHUNK_SIZE = 5 * 1024 * 1024; // 5MB
    let uploadedChunk = 0; // progress 추적하기 위한
    const fileName = createFileName(file.name);
    setFileNames([...prev, fileName]);
    try {
      const chunkedFile = await createChunkedArray(file, CHUNK_SIZE);
      const uploadId = await getIdForMultipartUpload(fileName, file.type);
      // 청크 배열을 순회하여 각 청크별로 업로드 요청 보냄
      const uploadPromises = chunkedFile.map(async (chunk, index) => {
        const uploadUrl = await getUploadUrlForChunk(
          fileName,
          index + 1,
          uploadId,
        );
        const response = await fetch(uploadUrl, {
          method: 'PUT',
          body: chunk,
          headers: {
            'Content-Type': file.type,
          },
        });
        uploadedChunk++;
        const progress = Math.ceil((uploadedChunk / chunkedFile.length) * 100);
        setUploadProgress(progress);
        if (!response.ok) throw new Error('Upload failed');
        return {
          Etag: response.headers.get('ETag') ?? '',
          PartNumber: index + 1,
        };
      });

      const completedParts: CompletedPartType[] =
        await Promise.all(uploadPromises);
      await closeMultipartUpload(fileName, uploadId, completedParts);

    } catch (error) {
      console.error('Upload error:', error);
      setIsUploading(false);
    } finally {
      setUploadProgress(0);
      setUploadedFileNumber((prev) => prev + 1);
    }
  }
  setIsUploading(false);
  setIsUploaded(true);
};
```

## 멀티파트 업로더 구현

AWS S3에 클라이언트에서 직접 파일을 업로드하여 서버의 부담을 줄였습니다.  
멀티파트 업로드를 위해, 직접 업로더를 구현하였습니다.

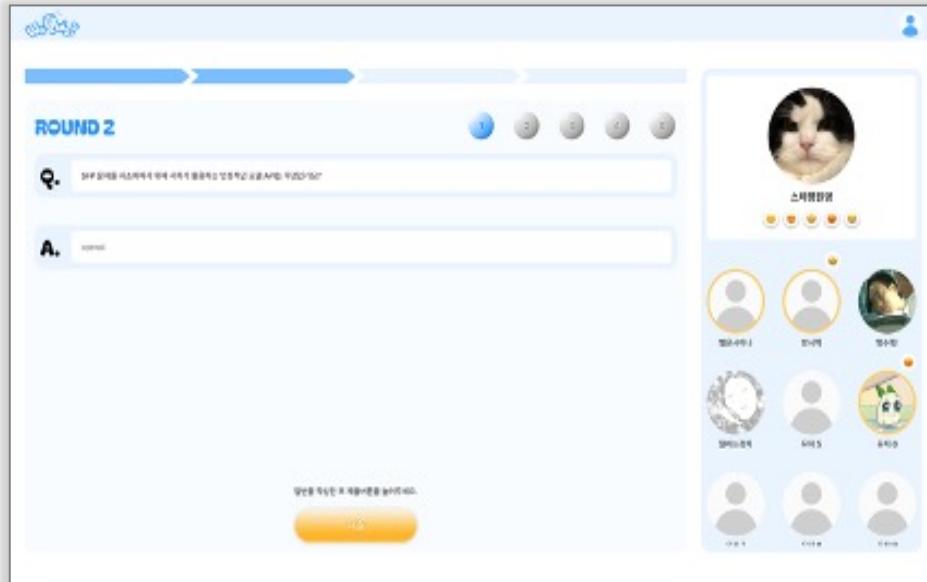
- 업로드된 파일을 5MB 단위의 파트로 분리합니다.
- 멀티파트 업로드 ID를 AWS S3 SDK를 활용하여 요청합니다.
- 각 파트에 대한 PresignedURL을 발급받습니다.
- PresignedURL로 \* 파트 수 만큼 요청을 보냅니다.
- PresignedURL 요청 수 / 총 파트 수를 계산하여 업로드 진행상황을 사용자에게 표시해줍니다.
- 모든 파트가 업로드 되었음을 S3 서버에 알리고, 스트림을 종료합니다.

업로더 구현을 통해 멀티파트 업로드 방식의 플로우를 깊게 이해할 수 있었고, 대용량 파일의 업로드 방식(파트 제한 5GB \* 파트 수 제한 10,000개 = 5TB)을 구현할 수 있게 되었습니다.

해당 기능은 추후 팀원과의 회의 끝에 업로드된 파일에 대한 책임을 백엔드에서 갖기로 변경하면서 deprecate 되었습니다.

### 3. 메인 작업

#### 문제 풀이 기능 (기여도 100%)



[문제 풀이 페이지]

#### 실시간 문제 풀이 기능 구현

소켓의 PUB/SUB 기능을 활용해 실시간 문제 풀이 기능을 구현하였습니다. 문제를 받아오고, 제출하고, 제출 상황을 사용자와 공유하는 기능 모두 소켓으로 구현하였습니다.

과반 수 이상의 참여자가 문제를 풀었을 경우, 3초의 카운트 후 자동으로 다음 문제로 넘어가게 됩니다.

스피드 퀴즈의 방식이기에, 사용자 경험 향상을 위해 엔터키를 입력하면 답변이 제출될 수 있도록 구현하였습니다. 답안을 작성하지 않았을 경우는 해당 기능을 사용할 수 없습니다.

백엔드 개발자와 협의하여 각 문제 풀이 흐름을 phase로 구분하고, 각 phase 별로 요청과 응답을 분기처리하여 사용자가 새로고침 시에도 게임을 유지할 수 있도록 구현하였습니다.

### 3. 메인 작업

## 문제 풀이 기능 (기여도 100%)

[BEFORE]

```
const [isLastQuiz, setIsLastQuiz] = useState(false);

const checkLastQuiz = () => {
  return quizSet?.quizNumber === (quizSet?.totalRound ?? 0) * 5;
}

const subscribeQuiz = (roomId: number) => {
  stompClient.subscribe(
    ...,
    setIsLastQuiz(checkLastQuiz()),
  ),
  ...
};

const subscribeScoreCount = (roomId: number) => {
  stompClient.subscribe(
    ...,
    if (countData.second === -1) {
      if (isLastQuiz) {
        closeModal();
        router.push(`/result/${params.id}`);
      }
    },
  );
};
```

[AFTER]

```
const lastQuizSetRef = useRef(lastQuizSet);
lastQuizSetRef.current = lastQuizSet;

useEffect(() => {
  if (quizSet?.quizNumber === (quizSet?.totalRound || 0) * 5) {
    const lastQuiz = quizSet;
    setLastQuizSet(lastQuiz);
  }
}, [quizSet]);

const subscribeScoreCount = (roomId: number) => {
  stompClient.subscribe(
    ...,
    if (countData.second === -1) {
      if (lastQuizSet) {
        router.push(`/result/${roomId}`);
      }
    }
);
```

### 상태 초기화 문제 해결

소켓을 활용하여 응답을 처리하는 과정에서 퀴즈의 상태를 가져오는 `getQuiz()`를 활용할 경우, 상태가 초기값으로 출력되는 문제가 발생했습니다. 커뮤니티를 서칭 해보았지만 비슷한 상황에 대한 해결책이 없었고, 문제 해결을 위해 자바스크립트 기본서를 다시 한 번 살펴 보았습니다. 이는 구독 시점의 상태가 이후 업데이트를 반영하지 못하는 closure 문제였고, 퀴즈 데이터에 `useRef`를 할당하고 구독 콜백 내에서 해당 ref를 참조하도록 하여 문제를 해결하였습니다.

이 트러블슈팅 경험을 통해 개념을 알고 있는 것과 실제 코드에 적용하는 것은 차이가 크다는 것을 깨달았습니다. 또한, 자바스크립트의 closure에 대해서 더 자세히 알게 되었으며, 라이브러리나 프레임워크를 사용하기 전에 자바스크립트의 기본기를 더 탄탄히 해야한다는 점을 다시 한 번 깨닫게 되었습니다.

### 3. 메인 작업

#### 결과 페이지 (기여도 100%)



[결과 페이지]

#### 결과 페이지 퍼블리싱 및 기능 구현

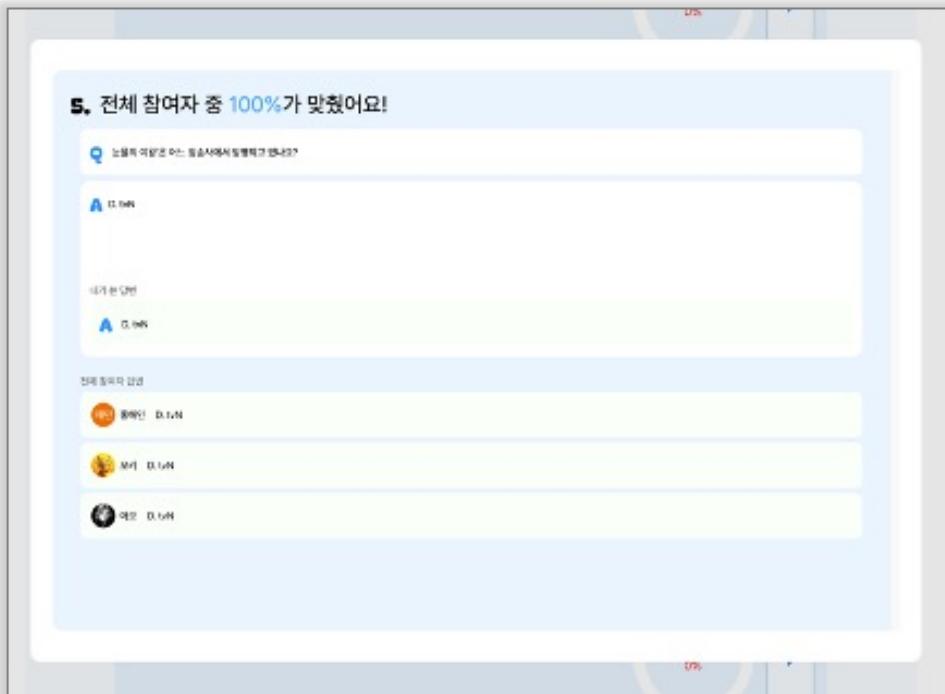
퀴즈가 모두 끝나면, 문제와 답변, 정답률을 확인할 수 있는 페이지를 구현하였습니다.

Flex를 기반으로 구현하였으며, 정답률 프로그레스 바는 큰 원과 작은 원을 겹치고, 정답률에 따라 큰 원의 conic-gradient를 조정하는 방식으로 구현하였습니다.

정답률에 따라 숫자와 원의 색깔을 변경함으로써 UI를 강화하였습니다.

### 3. 메인 작업

#### 상세 결과 모달 (기여도 100%)



[상세 결과 모달]

#### 상세 결과 모달 퍼블리싱 및 기능 구현

정답 페이지에서 상세보기 버튼을 클릭할 경우 띄워지는 상세 결과 모달을 구현하였습니다.

문제, 내가 쓴 답변, 정답, 전체 참여자의 답변을 확인할 수 있습니다.  
전체 답변자 컨테이너의 길이에 따라 부모 컨테이너의 overflowY: scroll을 적용하였습니다.

정답률에 따라 숫자의 색깔을 다르게 구분함으로써 UI를 강화하였습니다.

### 3. 메인 작업

#### 배경음악 (기여도 100%)

```
const useBgm = () => {
  const { setBgm } = useBgmStore();
  const url = usePathname();
  const quizUrl = ['/quiz-room'];
  const mainBgm = '/bgm/main.mp3';
  const quizBgm = '/bgm/quiz.mp3';
  const checkUrlAndChangeBgm = () => {
    if (quizUrl.includes(url)) {
      setBgm(quizBgm);
    } else {
      setBgm(mainBgm);
    }
  };
  return { checkUrlAndChangeBgm, url };
};
```

```
type BgmState = {
  bgm: string;
  isPlaying: boolean;
};

type BgmAction = {
  setIsPlaying: (isPlaying: boolean) => void;
  setBgm: (bgm: string) => void;
};

const useBgmStore = create<BgmState & BgmAction>((set) => ({
  bgm: '/bgm/main.mp3',
  isPlaying: false,
  setIsPlaying: (isPlaying) => set({ isPlaying }),
  setBgm: (bgm) => set({ bgm }),
}));
```

```
export default function BgmComponent() {
  const { bgm, isPlaying } = useBgmStore();
  const { checkUrlAndChangeBgm, url } = useBgm();
  const audioElement = useRef<HTMLAudioElement>(null);

  useEffect(() => {
    if (isPlaying) {
      audioElement.current?.play();
    } else {
      audioElement.current?.pause();
    }
  }, [isPlaying]);

  useEffect(() => {
    checkUrlAndChangeBgm();
  }, [url]);

  return <audio src={bgm} autoPlay loop ref={audioElement} />;
}
```

#### BGM 상태관리 및 컴포넌트, 커스텀 툴 제작

사용자의 bgm ON/OFF 여부, BGM 소스의 관리를 전역 상태로 구현하였습니다.

bgm의 설정을 위한 audio 태그만을 관리하는 컴포넌트를 제작하여 루트 레이아웃에 배치하였습니다.

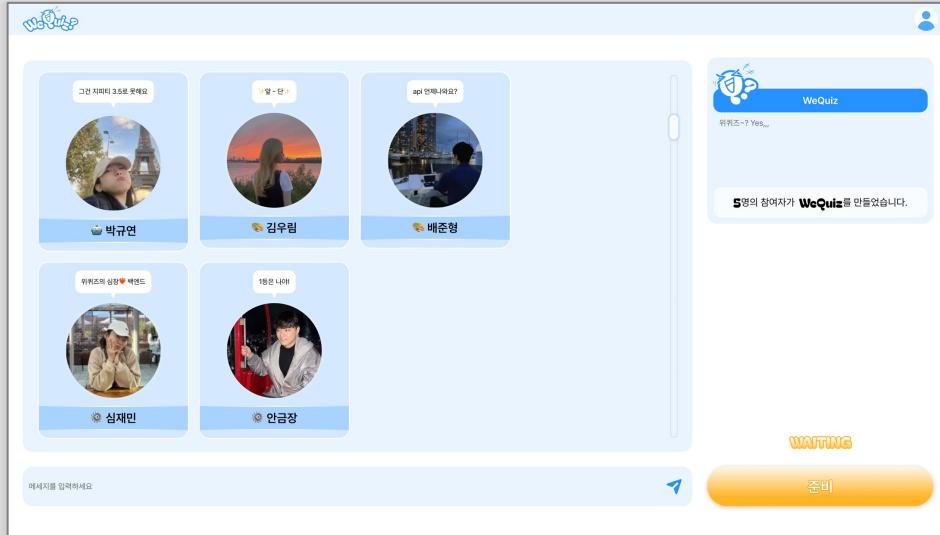
페이지 별로 다른 bgm을 재생하기 위해, 현재 접속한 페이지를 확인하고 페이지마다 할당된 bgm을 재생할 수 있도록 하는 커스텀 툴을 제작하여 관리하였습니다

#### 자동 재생 관련 문제

크롬, 파이어폭스, 사파리 등의 메이저 브라우저들은 대부분 UX를 위해 페이지에 입장하자마자 재생되는 미디어를 차단하고 있었다는 점을 확인했습니다. 이를 해결하기 위해, 초기 재생 여부를 false로 설정하고 초기 페이지에서 사용자가 시작 버튼을 누른 순간부터 bgm이 재생되도록 구현하였습니다.

### 3. 메인 작업

#### 대기 화면 (기여도 20%)



[대기실 페이지]

#### 크로스 브라우징 이슈 해결

모바일 환경에서 채팅을 입력하면, 마지막으로 입력했던 음절이 채팅창에 그대로 남아 있는 문제가 발생하였습니다. 이는 영어를 제외한 글자를 활용할 때의 IME 이슈로, OS단에서 IME는 자음/모음 조합을 위해 키 입력에 대한 이벤트 처리를 하는데 onKeyDown을 활용할 경우 브라우저에서도 키 입력에 대한 이벤트를 처리하게 되어 중복 입력이 발생한 것이 원인이었습니다. 클라이언트에서 키가 조합중임을 알리는 상태 IsComposition을 설정하여 composing 상태일 때 리턴하도록 처리하여 해결하였습니다.

### 3. 메인 작업

#### 성능 개선 작업

##### LCP 개선

모든 페이지에서 LCP 요소로 활용되는 메인 로고의 이미지 포맷을 webP로 변경하고, 이미지를 리사이즈 하였습니다. 또한 fetchPriority를 high로 조정하여 Render Delay를 1100ms -> 92ms로 약 **91.64%** 개선하였습니다.

##### 무한스크롤 개선

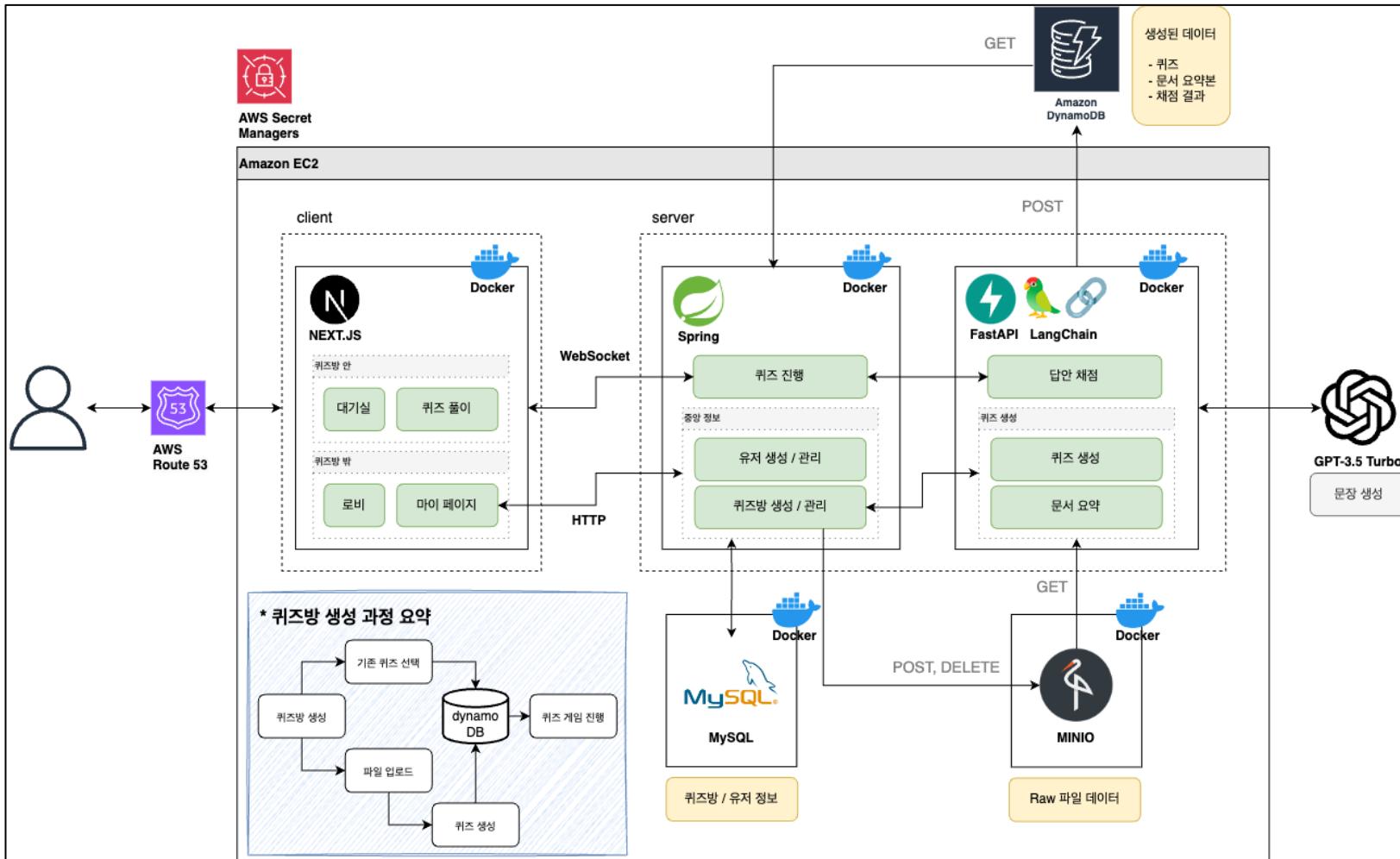
메인 로비의 방 목록을 커서 방식의 페이지네이션으로 활용하고 있었습니다. 더 미 데이터를 많이 쌓아 테스트 해 본 결과, 데이터가 커질 수록 프레임 드랍이 기하급수적으로 발생하는 것을 확인하였습니다. 이를 개선하기 위해 react-virtualized를 활용하여 부분 목록만 렌더링할 수 있도록 수정하였습니다. 그 결과 데이터의 크기에 상관없이 프레임 레이트를 60프레임으로 유지할 수 있었습니다. 이 기능은 추후 방 목록의 폴링 이슈로, 오프셋 방식의 페이지네이션으로 변경되어 폐지되었습니다.

##### 소켓 연결 개선

연결된 소켓이 장시간 활용되지 않았을 경우 소켓을 닫고, 요청이 오면 다시 여는 방식으로 구현하여 네트워크 자원 활용을 개선하였습니다.

## 4. 아키텍처

### 시스템 아키텍처



### 느낀 점(회고)

#### 실시간 통신 구현 경험

웹소켓을 활용하면서 실시간 데이터 전송과 처리에 대한 이해를 얻을 수 있었습니다. 이를 통해 실시간 애플리케이션의 복잡성을 알게 되었습니다. 특히, 서버와 클라이언트 간의 통신이 끊어졌을 때의 예외 처리를 경험하며 견고한 시스템을 구축하는 방법을 배웠습니다. 또한, 실시간으로 업데이트되는 점수와 제출상황을 구현하면서 UX의 중요함을 다시 한 번 깨닫게 되었습니다.

#### 효과적인 협업 도구의 사용

저를 포함한 5명의 팀원과 함께 협업을 진행하면서, 커뮤니케이션의 중요성에 대해 깨닫게 되었습니다. 프로젝트 초반에는 단순히 구두로 요구사항을 전달하고 진행 상황을 공유했으나, 이로 인한 오해가 많이 발생하였습니다. 이를 해결하기 위해 Storybook, Figma, Notion과 같은 협업 툴을 도입하였고, 시각적인 자료와 명확한 문서화를 통해 팀원 간의 이해를 높이고 커뮤니케이션의 질을 향상시키는데 큰 도움이 되었습니다.

#### 협업 및 커뮤니케이션

##### 팀원 간의 모멘텀 맞추기

백엔드 팀원 중 한 명은 이번 프로젝트로 백엔드를 처음 경험해 보는 것이라 프로젝트 시작 전에는 조금 걱정이 되었습니다. 하지만 결과적으로 그 팀원의 열정과 노력으로 프로젝트를 원활히 진행할 수 있었습니다. 이를 통해 현재의 실력보다 중요한 것은 팀원 간의 모멘텀을 맞추는 것이라는 깨달음을 얻었습니다. 모든 팀원이 같은 목표를 향해 노력하고, 서로를 지원하는 분위기가 팀의 성공에 핵심적임을 알게 되었습니다.

읽어 주셔서  
감사합니다

**배준형**

Tel: 010-7123-0537

Email: ryanbae94@gmail.com

Linkedin: <https://linkedin.com/in/junhyungbae>

Github: <https://github.com/ryanbae94>

Site: <https://ryanbae94.dev>