ChatGPT

Rolling API 엔드포인트 및 사용 가이드

Rolling API 소개 및 활용 흐름

Rolling API는 웹 롤링페이퍼 서비스를 위한 RESTful API입니다. 이를 통해 특정 **수신자**(받는 사람)를 생성하고, 여러 사람이 해당 수신자에게 **메시지**를 보내거나 보낸 메시지에 **반응(리액션)**을 달 수 있습니다. 프론트엔드 초보자도 쉽게 이해할 수 있도록, Rolling API의 주요 사용 흐름은 다음과 같습니다:

1. **수신자 생성:** 새로운 롤링페이퍼를 만들기 위해 **수신자**를 생성합니다. 예를 들어, 생일 축하 롤링페이퍼를 위해 수신자 이름을 등록합니다. (`POST /recipients`)
2. **메시지 보내기:** 생성된 수신자에게 다른 사람들이 축하 메시지나 응원 메시지를 작성하여 보냅니다. 각 메시지는 수신자 ID에 연결되어 저장됩니다. (`POST /messages` 등)
3. **메시지에 반응 달기:** 작성된 메시지를 보고 하트(❤) 등의 **반응**을 추가할 수 있습니다. 이를 통해 메시지에 공감 표시나 이모지 리액션을 달 수 있습니다. (`POST /reactions`)

이러한 흐름으로, **수신자 → 메시지 → 반응**의 순서로 Rolling API를 활용할 수 있습니다. 다음 섹션에서는 각 API 엔드포인트의 역할과 사용법을 자세히 살펴보겠습니다.

API 엔드포인트 요약

Rolling API에서는 롤링페이퍼 관련 기능을 수행하기 위해 여러 **엔드포인트**를 제공합니다. 각 엔드포인트는 `GET`, `POST`, `PUT`, `DELETE`와 같은 HTTP 메서드를 통해 데이터 조회, 생성, 수정, 삭제 등의 작업을 담당합니다. 아래 표는 주요 엔드포인트와 HTTP 요청별 기능을 요약한 것입니다:

엔드포인트	메서드	설명
`/recipients`	GET	모든 수신자 목록 조회 (관리용)
`/recipients`	POST	새로운 수신자 생성 (롤링페이퍼 개설)
`/recipients/{id}`	GET	특정 ID의 수신자 상세 조회
`/recipients/{id}`	PUT	특정 수신자 정보 수정 (예: 이름 변경)
`/recipients/{id}`	DELETE	특정 수신자 삭제
`/messages`	GET	조건에 따라 메시지 목록 조회[*]
`/messages`	POST	새로운 메시지 생성 (메시지 보내기)
`/messages/{id}`	GET	특정 메시지 상세 조회 (필요 시)
`/messages/{id}`	DELETE	특정 메시지 삭제 (관리자 또는 작성자 용)
`/reactions`	GET	조건에 따라 반응 목록 조회[*]
`/reactions`	POST	새로운 반응 추가 (리액션 달기)
`/reactions/{id}`	DELETE	특정 반응 삭제 (예: 반응 취소)

 `GET /messages`와 `GET /reactions`는 일반적으로 **쿼리 파라미터**를 사용하여 특정 수신자나 메시지에 관련된 항목만 조회합니다 (예: `GET /messages?recipientId={수신자ID}`는 해당 롤링페이퍼의 모든 메시지를 조회).*

위 엔드포인트들을 통해 프론트엔드에서는 순차적으로 **수신자를 생성**하고 → **메시지를 추가**하고 → **반응을 달고 관리**하는 기능을 구현할 수 있습니다. 다음으로, 각 엔드포인트별 상세 설명과 사용 예제를 Axios와 Fetch로 살펴보겠습니다.

수신자(Recipient) 엔드포인트 상세

수신자(Recipient) 엔드포인트는 롤링페이퍼를 받는 사람(또는 롤링페이퍼 자체)을 생성하고 관리하는 데 사용됩니다. 수신자는 롤링페이퍼의 **이름**(예: 이벤트명이나 받는 사람 이름 등 기본 식별 정보) 등을 포함합니다. 수신자 ID는 이후 메시지를 보낼 때 사용되며, 수신자를 생성함으로써 하나의 롤링페이퍼가 시작됩니다.

수신자 생성 – `POST /recipients`

이 요청은 새로운 수신자를 생성하여 롤링페이퍼를 개설합니다. **예시:** 사용자 입력으로 "홍길동 생일 축하"라는 이름의 수신자를 생성해보겠습니다.

- **요청 목적:** 새로운 수신자 리소스 생성 (롤링페이퍼 시작). 서버는 생성된 수신자의 고유 ID와 정보를 응답합니다.
- **요청 본문:** JSON 형식으로 수신자 정보 전달. 최소 필드는 `name` (수신자 이름)이며, 필요 시 기타 설명 필드가 있을 수 있습니다.
- **응답:** 성공 시 생성된 **Recipient 객체**가 반환되며, 여기에는 고유 `id`, `name`, 생성 시각 등의 필드가 포함됩니다.

요청에 사용할 데이터 타입 **RecipientCreate** (Swagger 정의된 가칭) 구조:

```
{
 "name": "홍길동"  // 수신자 이름 (필수)
}
```

성공 시 반환되는 **Recipient** 객체 예:

```
{
 "id": 1,
 "name": "홍길동",
 "createdAt": "2025-06-02T02:32:22.000Z"
}
```

(예시 JSON - 실제 `createdAt` 등의 필드는 구현에 따라 다를 수 있습니다.)

이제 React 컴포넌트에서 **Axios**와 **Fetch**를 사용하여 수신자를 생성하는 코드를 살펴보겠습니다.

Axios 사용 예제:

```
// 수신자 생성 버튼 클릭 시 실행되는 핸들러 (Axios 사용)
import axios from 'axios';

function createRecipient(name) {
 const API_URL = 'https://rolling-api.vercel.app/recipients';
 axios.post(API_URL, { name })  // POST 요청으로 name 전달
  .then(response => {
   // 성공 시, response.data에 생성된 수신자 객체가 들어있습니다.
   console.log('생성된 수신자:', response.data);
   // TODO: 추가 동작 (예: 생성된 ID로 메시지 작성 페이지로 이동)
  })
  .catch(error => {
   // 오류 처리: 유효성 검증 실패 또는 네트워크 오류 등
   if (error.response) {
    // 서버가 오류 응답을 보낸 경우 (예: 400 Bad Request)
    alert(`생성 실패: ${error.response.data.message}`);
   } else {
    // 그 외 네트워크 오류 등
    alert('수신자 생성 중 오류가 발생했습니다.');
   }
  });
}
```

Fetch 사용 예제:

```
// 수신자 생성 버튼 클릭 시 실행되는 핸들러 (Fetch 사용)
function createRecipient(name) {
 const API_URL = 'https://rolling-api.vercel.app/recipients';
 fetch(API_URL, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ name })  // 요청 본문에 JSON 문자열로 name 전달
 })
  .then(response => {
   if (!response.ok) {
    // HTTP 오류 상태 처리 (response.ok가 false인 경우)
    return response.json().then(err => {
     throw new Error(err.message || '수신자 생성 실패');
    });
   }
   return response.json();
  })
  .then(data => {
   console.log('생성된 수신자:', data);
   // TODO: 추가 동작 (예: 메시지 작성 페이지로 이동 등)
  })
  .catch(error => {
   alert(`수신자 생성 중 오류: ${error.message}`);
```

```
    });
}
```

위의 코드에서 Axios와 Fetch 모두 **POST** 요청을 통해 `/recipients` 엔드포인트를 호출하고 있습니다. `axios.post`는 자동으로 JSON을 직렬화하며, Fetch를 사용할 때는 `headers`와 `JSON.stringify`를 수동으로 지정해야 합니다. 또한, **오류 처리 방식**이 약간 다른 점에 유의하세요: Axios는 HTTP 오류 시 자동으로 catch로 넘어가지만, Fetch는 `.then` 단계에서 `response.ok`를 직접 확인하여 오류를 처리해야 합니다.

📋 수신자 목록 조회 – `GET /recipients`

(※ 일반 사용자 시나리오에서는 자주 쓰이지 않을 수 있지만, 관리 용도나 테스트를 위해 제공됩니다.)

- **요청 목적:** 서버에 저장된 **모든 수신자 목록**을 가져옵니다. (관리자나 디버깅 용도로 사용)
- **요청 본문:** 없음. (`GET` 요청)
- **응답:** 수신자 객체들의 배열이 반환됩니다.

예시 응답:

```
[
 {
  "id": 1,
  "name": "홍길동",
  "createdAt": "2025-06-02T02:32:22.000Z"
 },
 {
  "id": 2,
  "name": "김철수",
  "createdAt": "2025-06-03T10:15:00.000Z"
 }
]
```

Axios 사용 예제:

```
// 컴포넌트가 마운트될 때 전체 수신자 목록을 조회 (Axios)
useEffect(() => {
 axios.get('https://rolling-api.vercel.app/recipients')
  .then(response => {
   setRecipients(response.data); // 수신자 목록 상태 업데이트
  })
  .catch(error => {
   console.error('수신자 목록 조회 오류:', error);
  });
}, []);
```

Fetch 사용 예제:

```
// 컴포넌트가 마운트될 때 전체 수신자 목록을 조회 (Fetch)
useEffect(() => {
 fetch('https://rolling-api.vercel.app/recipients')
  .then(response => response.json())
  .then(data => {
   setRecipients(data); // 수신자 목록 상태 업데이트
  })
  .catch(error => {
   console.error('수신자 목록 조회 오류:', error);
  });
}, []);
```

위 예제는 컴포넌트가 처음 렌더링될 때(`useEffect` 활용) 수신자 목록을 가져오는 코드입니다. Axios의 응답 데이터는 `response.data`에 담기며, Fetch는 별도의 `.then(response => response.json())` 단계를 거칩니다.

🔍 수신자 상세 조회 – `GET /recipients/{id}`

특정 수신자 ID에 대한 정보를 조회합니다. 보통 개별 롤링페이퍼의 제목이나 수신자 이름을 표시하거나, 존재 여부를 확인하는 데 사용됩니다.

- **요청 목적:** 특정 **id**를 가진 수신자의 상세 정보 확인.
- **요청 경로:** `/recipients/수신자ID` 예를 들어 `/recipients/1`이면 ID가 1인 수신자 조회.
- **응답:** 해당 수신자에 대한 **Recipient 객체**를 반환. (존재하지 않는 ID의 경우 404 에러 응답)

사용 예시 (Fetch):

```
// 특정 수신자 ID의 정보를 가져오기 (예: id가 1인 수신자)
fetch(`https://rolling-api.vercel.app/recipients/1`)
 .then(res => res.json())
 .then(data => {
  console.log('수신자 1의 정보:', data);
  // data 예시: { id: 1, name: '홍길동', createdAt: '...'}
 })
 .catch(err => console.error('상세 조회 오류:', err));
```

(Axios 사용 시에는 `axios.get('/recipients/1').then(response => ...)`와 비슷하게 사용하면 됩니다.)

✏️ 수신자 수정 – `PUT /recipients/{id}`

이미 생성된 수신자의 정보를 수정합니다. 일반적으로 수신자의 이름이나 기타 속성을 변경할 때 사용됩니다.

- **요청 목적:** 수신자의 정보 업데이트 (예: 이름을 잘못 입력한 경우 수정).
- **요청 경로 및 본문:** `/recipients/{id}`에 대해 **PUT** 메서드 사용. 본문에 수정할 필드를 JSON으로 포함 (예: `{ "name": "새 이름" }`).
- **응답:** 수정된 결과의 **Recipient 객체**를 반환하거나 상태 코드 204 (콘텐츠 없음) 등을 반환할 수 있습니다. (API 구현에 따라 다르지만, 여기서는 편의상 수정된 객체를 돌려준다고 가정)

Axios 사용 예제:

```
// 수신자 이름 수정 (id=1인 수신자의 이름을 "김철수"로 변경)
axios.put('https://rolling-api.vercel.app/recipients/1', { name: '김철수' })
  .then(res => {
    console.log('수정된 수신자 정보:', res.data);
    // 예: { id: 1, name: '김철수', createdAt: '...', ... }
  })
  .catch(err => {
    console.error('수신자 수정 오류:', err);
  });
```

🗑️ 수신자 삭제 – `DELETE /recipients/{id}`

특정 수신자를 삭제합니다. **주의:** 수신자를 삭제하면 관련된 모든 메시지와 반응 데이터도 함께 삭제될 수 있습니다.

- **요청 목적:** 더 이상 필요 없는 롤링페이퍼(수신자)를 삭제.
- **요청 경로:** `/recipients/{id}`에 대해 **DELETE** 메서드 사용.
- **응답:** 일반적으로 성공 시 **204 No Content** 상태 코드가 반환됩니다. (별도의 응답 본문 없음)

Fetch 사용 예제:

```
// id=1인 수신자 삭제 요청
fetch('https://rolling-api.vercel.app/recipients/1', { method: 'DELETE' })
  .then(response => {
    if (response.ok) {
      console.log('수신자 삭제 성공');
    } else {
      console.error('삭제 실패: 상태 코드', response.status);
    }
  })
  .catch(err => {
    console.error('수신자 삭제 요청 오류:', err);
  });
```

수신자 API를 통해 새로운 롤링페이퍼를 만들고 관리할 수 있습니다. 이제 **메시지 API**를 사용하여 롤링페이퍼에 메시지를 추가하고 조회하는 방법을 알아보겠습니다.

메시지(Message) 엔드포인트 상세

메시지(Message) 엔드포인트는 특정 수신자(롤링페이퍼)에 담기는 **메시지**들을 생성, 조회, 삭제하는데 사용됩니다. 여기서 메시지는 롤링페이퍼에 남기는 개별 글(예: "생일 축하해!", "항상 고마워" 등의 문구)을 말합니다. 메시지에는 보통 **내용(content)**과 **작성자명(익명으로 남길 경우 닉네임 등)**을 포함할 수 있으며, 어떤 수신자에게 보내는지에 대한 **수신자 ID** 정보가 필요합니다.

메시지 보내기(생성) – `POST /messages`

이 요청은 특정 롤링페이퍼에 새로운 메시지를 작성하여 추가합니다. **수신자 ID**를 지정해야 하며, 메시지 **내용**과 (선택적으로) 작성자 이름이나 별명을 함께 보낼 수 있습니다.

- **요청 목적:** 선택된 롤링페이퍼(수신자)에 메시지를 추가(생성).
- **요청 본문: MessageCreate** 데이터 구조를 JSON으로 전달. 예시 구조:

```
{
  "recipientId": 1,      // 메시지를 보낼 대상 수신자의 ID
  "content": "생일 진심으로 축하해!",  // 메시지 내용
  "senderName": "철수"     // (선택) 작성자 이름 또는 별명
}
```

- `recipientId` (number): 반드시 존재해야 하는 필드입니다. 어느 수신자(롤링페이퍼)에 속할 메시지인지 지정합니다.
- `content` (string): 메시지 본문 텍스트. (필수, 내용이 비어 있으면 서버에서 오류 응답 가능)
- `senderName` (string, 선택): 작성자의 이름 또는 표시할 닉네임. 제공하지 않으면 기본값(예: "익명")으로 처리될 수 있습니다.

- **응답:** 성공 시 생성된 **Message** 객체를 반환합니다. 이 객체에는 고유 `id`, `content`, `senderName`, `recipientId`, `createdAt` 등의 필드가 포함됩니다.

예시 응답 객체:

```
{
  "id": 101,
  "recipientId": 1,
  "content": "생일 진심으로 축하해!",
  "senderName": "철수",
  "createdAt": "2025-06-02T03:00:00.000Z"
}
```

Axios 사용 예제:

```
// 메시지 작성 폼 제출 시 호출 (Axios 사용)
import axios from 'axios';

function sendMessage(recipientId, content, senderName) {
  axios.post('https://rolling-api.vercel.app/messages', {
    recipientId,
    content,
    senderName
  })
  .then(res => {
    console.log('메시지 전송 성공:', res.data);
    // TODO: 전송한 메시지를 화면에 추가하거나 알림 표시
```

```
  })
  .catch(err => {
   if (err.response) {
    // 서버에서 오류 응답을 준 경우 (예: 필드 누락 등의 400 에러)
    alert(`메시지 전송 실패: ${err.response.data.message}`);
   } else {
    // 네트워크 등 기타 오류
    alert('메시지 전송 중 문제가 발생했습니다.');
   }
  });
}
```

Fetch 사용 예제:

```
// 메시지 작성 폼 제출 시 호출 (Fetch 사용)
function sendMessage(recipientId, content, senderName) {
 fetch('https://rolling-api.vercel.app/messages', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ recipientId, content, senderName })
 })
  .then(response => {
   if (!response.ok) {
    // 서버 오류 발생 시
    return response.json().then(err => {
     throw new Error(err.message || '메시지 전송 실패');
    });
   }
   return response.json();
  })
  .then(data => {
   console.log('메시지 전송 성공:', data);
   // TODO: 전송된 메시지를 화면에 추가
  })
  .catch(error => {
   alert(`메시지 전송 중 오류: ${error.message}`);
  });
}
```

위 예제에서는 `recipientId`로 지정된 롤링페이퍼에 메시지를 전송합니다. **유효성 검사**로 `content`가 비어있거나 너무 길면 **400 Bad Request** 등의 오류가 발생할 수 있으며, 이러한 경우 `.catch`에서 오류를 처리하고 사용자에게 안내합니다.

📩 메시지 목록 조회 – `GET /messages?recipientId={id}`

특정 수신자(롤링페이퍼)에 달린 **모든 메시지 목록**을 불러옵니다. 이 엔드포인트는 주로 롤링페이퍼 화면에서 현재까지 작성된 메시지들을 표시하기 위해 사용됩니다.

- **요청 목적:** 특정 수신자 ID와 연관된 메시지들을 모두 조회.

- **요청 방식:** `GET /messages?recipientId=수신자ID` 형식으로 **쿼리 파라미터** 사용. 예: `/messages?recipientId=1`은 ID가 1인 롤링페이퍼의 메시지 전체 목록을 가져옵니다.
- **요청 본문:** 없음. (URL에 쿼리만 지정)
- **응답:** 메시지 객체들의 배열을 반환. 각 메시지 객체에는 `id`, `content`, `senderName`, `recipientId`, `createdAt` 등이 포함됩니다. 서버는 기본적으로 `recipientId`로 필터링된 메시지들만 리턴합니다.

예시 응답:

```
[
 {
  "id": 101,
  "recipientId": 1,
  "content": "생일 진심으로 축하해!",
  "senderName": "철수",
  "createdAt": "2025-06-02T03:00:00.000Z"
 },
 {
  "id": 102,
  "recipientId": 1,
  "content": "Always stay happy and healthy   ",
  "senderName": "Anna",
  "createdAt": "2025-06-02T03:05:10.000Z"
 }
]
```

Fetch 사용 예제 (React 컴포넌트에서 메시지 목록 가져오기):

```
// recipientId에 해당하는 메시지 목록을 가져와 상태에 저장
useEffect(() => {
 const recipientId = 1;  // 예: 현재 보고있는 롤링페이퍼 ID
 fetch(`https://rolling-api.vercel.app/messages?recipientId=${recipientId}`)
  .then(res => res.json())
  .then(list => {
   setMessageList(list);  // 메시지 목록 상태 업데이트
  })
  .catch(err => {
   console.error('메시지 목록 불러오기 실패:', err);
  });
}, []);
```

(참고: Axios를 사용하면 `axios.get('/messages', { params: { recipientId } })`와 같이 쿼리 파라미터를 전달할 수 있습니다. 예: `axios.get('https://rolling-api.vercel.app/messages', { params: { recipientId: 1 } })`)

이렇게 조회한 메시지 목록 데이터를 화면에 뿌려주어 롤링페이퍼에 달린 모든 메시지를 사용자에게 보여줄 수 있습니다.

메시지 수정 – `PUT /messages/{id}`

Note: 일반적인 롤링페이퍼 시나리오에서는 한 번 작성한 메시지를 수정하지 않고 그대로 두는 경우가 많습니다. 그러나 API가 제공된다면 메시지 내용을 편집할 수도 있습니다.

- **요청 목적:** 특정 메시지의 내용을 수정 (예: 오타를 고치거나 내용을 변경).
- **요청 경로:** `/messages/{메시지ID}`에 대해 **PUT**. 본문에는 수정할 필드(예: `content`)를 JSON으로 포함. (`recipientId`나 `senderName`도 변경 가능하게 구현되었을 수 있음)
- **응답:** 수정된 **Message 객체**를 반환하거나 상태 코드 204를 반환.

Axios 사용 예제:

```
// id=101인 메시지의 내용을 수정 (예: 내용 변경)
axios.put('https://rolling-api.vercel.app/messages/101', {
  content: "내용을 수정했습니다."
 })
 .then(res => {
  console.log('메시지 수정 성공:', res.data);
 })
 .catch(err => {
  console.error('메시지 수정 오류:', err);
 });
```

메시지 삭제 – `DELETE /messages/{id}`

특정 메시지를 삭제합니다. 잘못된 메시지를 지우거나 부적절한 내용이 있을 때 사용할 수 있습니다.

- **요청 목적:** 메시지 삭제 (수신자 관리자가 전체 삭제를 허용하거나, 작성자 본인이 삭제하는 시나리오 등).
- **요청 경로:** `/messages/{메시지ID}`에 **DELETE** 메서드 사용.
- **응답:** 성공 시 204 No Content (혹은 삭제 결과에 대한 간단한 메시지를 줄 수 있습니다).

Fetch 사용 예제:

```
// 특정 메시지 삭제 요청 (예: id=101 메시지 삭제)
fetch('https://rolling-api.vercel.app/messages/101', { method: 'DELETE' })
 .then(response => {
  if (response.ok) {
   // 메시지 리스트 상태에서 해당 메시지 제거 등 후속 처리
   console.log('메시지 삭제됨');
  } else {
   console.error('메시지 삭제 실패:', response.status);
  }
 })
 .catch(err => {
  console.error('삭제 요청 오류:', err);
 });
```

이상으로 메시지 생성, 조회, 수정, 삭제 기능을 모두 살펴보았습니다. 다음은 **반응(리액션) API**를 알아보겠습니다.

반응(Reaction) 엔드포인트 상세

반응(Reaction) 엔드포인트는 작성된 메시지에 **이모지 반응**(리액션)을 추가하거나 제거하는 데 사용됩니다. 예를 들어 누군가의 메시지에 하트 표시를 눌러 공감 표시를 하는 기능입니다. 보통 사용자 로그인이나 고유 식별이 없다면 한 사람당 여러 번 누를 수 있으므로, 단순히 **카운트 증가** 개념으로 구현되기도 합니다. 여기서는 각 반응을 하나의 리소스로 보고 추가/삭제할 수 있는 API로 설명합니다.

👍 반응 추가 – `POST /reactions`

특정 메시지에 새로운 반응을 추가합니다. 예를 들어 메시지 ID 101에 하트(❤) 반응을 달 수 있습니다.

- **요청 목적:** 메시지에 리액션(이모지 반응)을 **추가**.
- **요청 본문:** **ReactionCreate** 구조를 JSON으로 전달. 예시:

```
{
  "messageId": 101,   // 어느 메시지에 대한 반응인지 지정
  "emoji": "heart"    // 반응의 종류 (예: heart, thumbs_up 등의 식별자 또는 이모지 문자)
}
```

- `messageId` (number): 반드시 포함. 어떤 메시지에 반응을 다는지 식별합니다.
- `emoji` (string): 추가할 반응의 종류나 이모지. (API에서 지원하는 특정 키워드나 이모지 문자열을 사용)
- **응답:** 성공 시 추가된 **Reaction 객체**를 반환하거나, 해당 메시지의 최신 반응 목록/개수를 반환할 수 있습니다. Reaction 객체에는 보통 `id`, `messageId`, `emoji`, `createdAt` 등이 포함됩니다. (특정 사용자 식별이 있다면 그 정보도 포함되겠지만, 여기서는 없다 가정)

예시 응답:

```
{
  "id": 55,
  "messageId": 101,
  "emoji": "heart",
  "createdAt": "2025-06-02T04:00:00.000Z"
}
```

(이 응답은 새 반응 하나에 대한 객체 예시입니다. 경우에 따라 서버는 단순히 성공 상태만 돌려주고, 별도 데이터는 제공하지 않을 수도 있습니다.)

Axios 사용 예제:

```
// 메시지에 하트 반응 달기 (Axios 사용)
axios.post('https://rolling-api.vercel.app/reactions', {
    messageId: 101,
    emoji: 'heart'
  })
  .then(res => {
```

```
  console.log('반응 추가 성공:', res.data);
  // TODO: UI에서 해당 메시지의 하트 개수 증가 처리 등
 })
 .catch(err => {
  console.error('반응 추가 오류:', err);
 });
```

Fetch 사용 예제:

```
// 메시지에 하트 반응 달기 (Fetch 사용)
fetch('https://rolling-api.vercel.app/reactions', {
 method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify({ messageId: 101, emoji: 'heart' })
})
 .then(response => response.json())
 .then(data => {
  console.log('반응 추가 성공:', data);
  // TODO: UI 갱신 처리
 })
 .catch(err => {
  console.error('반응 추가 실패:', err);
 });
```

반응 추가 후에는 화면상의 해당 **메시지에 대한 반응 수**나 아이콘 표시를 업데이트해야 할 것입니다. 응답으로 새 반응 객체나 전체 반응 리스트를 받았다면 이를 활용하면 됩니다.

📊 반응 목록 조회 – `GET /reactions?messageId={id}`

특정 메시지에 달린 모든 반응을 조회합니다. (이 기능이 필요하지 않을 수도 있지만, 예를 들어 좋아요 수 등을 새로고침 없이 가져오거나, 여러 종류의 이모지를 지원한다면 각각 몇 개 달렸는지 보여줄 때 쓰일 수 있습니다.)

- **요청 목적:** 특정 메시지에 달린 모든 반응(들)의 정보를 가져옴.
- **요청 방식:** `GET /reactions?messageId=메시지ID`. 예: `/reactions?messageId=101`은 메시지 101의 모든 반응 리스트를 반환.
- **응답:** 해당 메시지의 Reaction 객체 배열. (혹은 간단히 개수만 응답할 수도 있음. API 구현에 따라 다름)

예시 응답:

```
[
 { "id": 55, "messageId": 101, "emoji": "heart", "createdAt": "2025-06-02T04:00:00.000Z" },
 { "id": 56, "messageId": 101, "emoji": "heart", "createdAt": "2025-06-02T04:05:00.000Z" },
 { "id": 57, "messageId": 101, "emoji": "thumbs_up", "createdAt": "2025-06-02T05:00:00.000Z" }
]
```

위 예시는 메시지 101에 heart 반응 2개와 thumbs_up 반응 1개가 달린 경우입니다. (만약 같은 유형의 이모지 반응을 서버에서 합쳐서 관리한다면, 응답 구조는 다를 수 있지만 여기서는 단순 리스트로 가정합니다.)

반응 삭제 – `DELETE /reactions/{id}`

특정 반응을 제거합니다. 예를 들어 사용자가 잘못 눌렀거나, 반응 취소 기능이 있는 경우 사용됩니다.

- **요청 목적:** 반응 하나 삭제 (보통 해당 반응 ID를 알고 있어야 함).
- **요청 경로:** `/reactions/{반응ID}`에 **DELETE** 요청.
- **응답:** 성공 시 204 No Content.

Axios 사용 예제:

```
// id=55인 반응(heart 하나)을 삭제
axios.delete('https://rolling-api.vercel.app/reactions/55')
 .then(() => {
  console.log('반응 삭제 성공');
  // TODO: UI에서 해당 반응 제거 또는 개수 감소 처리
 })
 .catch(err => {
  console.error('반응 삭제 오류:', err);
 });
```

> **참고:** 반응(리액션)은 일반적인 소셜 서비스에서는 사용자마다 한 번씩만 누를 수 있게 하거나 중복을 막지만, Rolling API에서는 별도의 사용자 인증이 없는 경우 단순히 엔트리를 추가하는 방식으로 구현될 수 있습니다. 따라서 같은 사람이 여러 번 추가하면 리스트에 여러 항목이 생길 수 있습니다. 프론트엔드에서는 필요에 따라 중복 추가를 막거나, 한 종류의 이모지별 개수만 보여주는 등의 처리를 할 수 있습니다.

이상으로 Rolling API의 주요 엔드포인트 (수신자, 메시지, 반응)에 대한 설명과 활용 예제를 모두 살펴보았습니다. 다음은 요청과 응답 데이터의 구조를 정리하고, API 사용 시 발생할 수 있는 오류 상황과 처리 방법에 대해 알아보겠습니다.

요청 및 응답 데이터 타입 구조

이 섹션에서는 앞서 사용된 **요청(Request)과 응답(Response)에 등장하는 데이터 타입**들의 필드 구조를 정리합니다. (Swagger 문서에 정의된 내용을 바탕으로 작성한 가이드입니다.)

Recipient (수신자) 객체

필드명	타입	설명
id	number	수신자의 고유 ID (자동 생성)
name	string	수신자 이름 (롤링페이퍼 제목이나 대상 이름)
createdAt	string(DateTime)	수신자 생성 시각 (ISO 날짜 문자열 등)
...	...	(추가 필드가 있을 경우 이어서)

- **RecipientCreate 요청 구조:** `{ "name": "홍길동" }` 등 **수신자 생성**에 필요한 필드만 포함 (현재는 `name`만 필요).

Message (메시지) 객체

필드명	타입	설명
id	number	메시지의 고유 ID (자동 생성)
recipientId	number	이 메시지가 속한 수신자(롤링페이퍼)의 ID
content	string	메시지 내용
senderName	string	작성자 이름 또는 닉네임 (익명 가능)
createdAt	string(DateTime)	메시지 작성 시각
...	...	(추가 필드가 있을 경우 이어서)

- **MessageCreate 요청 구조:** `{ "recipientId": 1, "content": "메시지 내용", "senderName": "이름" }` 형태. `recipientId`와 `content`는 필수이며, `senderName`는 선택일 수 있습니다.

Reaction (반응) 객체

필드명	타입	설명
id	number	반응(리액션)의 고유 ID (자동 생성)
messageId	number	이 반응이 달린 대상 메시지의 ID
emoji	string	반응 종류를 나타내는 식별자 또는 이모지
createdAt	string(DateTime)	반응이 추가된 시각
...	...	(추가 필드가 있을 경우 이어서)

- **ReactionCreate 요청 구조:** `{ "messageId": 101, "emoji": "heart" }` 형태로, `messageId`와 `emoji` 필드를 포함합니다.

以上 필드 구조는 API의 Swagger 문서를 기반으로 한 예시이며, 실제 구현에 따라 일부 필드나 명칭이 다를 수 있습니다. 하지만 전반적으로 **Recipient**는 롤링페이퍼(방)를 식별하는 정보, **Message**는 개별 메시지의 내용과 작성자 정보, **Reaction**은 메시지에 대한 특정 반응 정보로 구성됩니다.

오류 발생 상황과 처리 방법

Rolling API를 연동하면서 발생할 수 있는 **오류 상황**과 이에 대한 처리 방법을 정리합니다. 프론트엔드에서는 네트워크 통신 중 오류가 발생하더라도 앱이 중단되지 않고 사용자에게 적절한 피드백을 주어야 합니다.

네트워크 오류

- **설명:** 사용자의 인터넷 연결 문제, 서버 다운, 혹은 CORS 설정 오류 등으로 요청 자체가 실패하는 경우 발생합니다.
- **특징:** 이 경우 서버로부터 응답이 없으므로 `axios`의 `error.response`가 정의되지 않거나, `fetch`의 `catch`로 바로 들어갑니다.
- **처리:**
- Axios의 경우 `error.response`를 확인하여 존재하지 않으면 네트워크 오류로 간주하고 안내 메시지를 표시합니다.

- Fetch의 경우 `.catch`에서 잡힌 `TypeError` 등을 네트워크 오류로 판단할 수 있습니다.
- 사용자에게 "네트워크 연결에 문제가 있습니다. 인터넷 연결을 확인해주세요." 등 안내.

예시 (Axios):

```
axios.get('/some-url')
 .catch(error => {
  if (!error.response) {
   alert('요청 실패: 네트워크 오류가 발생했습니다.');
  }
 });
```

⚠ 유효성 검사 오류 (잘못된 요청 데이터)

- **설명:** 요청 본문에 필요한 필드가 누락되었거나 형식이 잘못된 경우, 서버는 **400 Bad Request** 등의 응답과 함께 오류 메시지를 보냅니다. 예를 들어 `POST /messages`에 `content`가 빈 문자열로 전송되면 유효성 검증에 걸릴 수 있습니다.
- **특징:** Axios에서는 `.catch`로 들어오며 `error.response.status`가 400 범위 (또는 422 등) 일 수 있고, `error.response.data`에 자세한 오류 내용이 담겨 있을 수 있습니다. Fetch의 경우 `.then`의 `response.ok`가 `false`가 되어 직접 에러 메시지를 추출해야 합니다.
- **처리:**
- 서버가 준 오류 메시지를 사용자에게 보여주거나 (`error.response.data.message` 등을 사용) 폼 필드 옆에 표시하여 무엇이 잘못되었는지 안내합니다.
- 예: "메시지 내용은 필수 입력 항목입니다." 등의 구체적인 피드백 제공.

예시 (Fetch):

```
fetch('/messages', { method: 'POST', /* ... */ })
 .then(response => {
  if (!response.ok) {
   return response.json().then(errData => {
    // 서버가 보낸 에러 메시지가 있으면 추출
    const msg = errData.message || '요청 데이터 오류';
    throw new Error(msg);
   });
  }
  return response.json();
 })
 .catch(error => {
  alert(`요청 실패: ${error.message}`);
 });
```

인증 및 권한 오류

- **설명:** (Rolling API가 별도의 인증을 요구하지 않는 공개 API라면 해당 없음으로 볼 수 있으나,) 만약 특정 엔드포인트가 인증이 필요하거나 권한이 있어야 한다면 **401 Unauthorized** 또는 **403 Forbidden** 오류가 발생할 수 있습니다.

- **처리:** 이 경우 로그인 페이지로 리다이렉트하거나, "접근 권한이 없습니다."라는 메시지를 보여줍니다. Rolling API에서는 일반 사용 시 인증이 없다고 가정하지만, 수신자 삭제 등은 내부적으로 보호되어 있을 수 있습니다.

🔍 리소스 미존재 오류 (404 Not Found)

- **설명:** 요청한 리소스(ID 등)가 존재하지 않는 경우입니다. 예를 들어 잘못된 수신자 ID로 `GET /recipients/999`를 호출하면 404 응답.
- **처리:** 사용자가 요청한 대상이 없음을 알리고, 화면에서는 "존재하지 않는 데이터입니다." 또는 필요 시 해당 화면에서 벗어나도록 처리합니다.

서버 오류 (500 Internal Server Error 등)

- **설명:** 서버측 코드 실행 중 예외가 발생한 경우 등입니다.
- **처리:** 사용자에게 명확한 원인을 노출하지 않는 것이 일반적입니다. "서버 오류가 발생했습니다. 잠시 후 다시 시도해주세요."와 같이 안내하고, 필요하면 오류 보고를 위해 로그를 남깁니다.

✔ 오류 처리 모범 사례

- **일괄 처리:** Axios 인터셉터나 Fetch 공통 함수 등을 활용해 모든 요청의 오류를 중앙에서 처리하는 것도 방법입니다. 그러나 초보자의 경우 각 요청별 `.catch`에서 개별 처리하는 것이 이해하기 쉬울 수 있습니다.
- **사용자 피드백:** 오류 발생 시 `alert`이나 페이지 상의 UI 요소로 즉시 알려주는 것이 좋습니다. 특히 입력 폼 오류의 경우 단순 실패 알림보다 어떤 필드가 문제인지 하이라이트해주는 것이 UX에 도움이 됩니다.
- **콘솔 로그:** 개발 중에는 `console.error`를 통해 오류 객체를 출력해보면 문제가 되는 부분을 파악하는 데 유용합니다. 배포 후에는 필요한 경우를 제외하고 콘솔 로그를 지우거나, Sentry와 같은 도구로 수집하는 방안을 고려합니다.

마치며 (정리)

이상으로 **Rolling API**의 엔드포인트별 사용 방법을 Axios와 Fetch 예제 코드와 함께 살펴보았습니다. 프론트엔드 초보자라도 위 가이드를 통해 다음을 이해할 수 있었을 것입니다:

- 각 엔드포인트의 역할과 어떤 상황에 어떤 HTTP 메서드를 써야 하는지 (예: 생성은 POST, 조회는 GET 등).
- React 컴포넌트 내에서 API를 호출하는 패턴 (`useEffect`로 초기 데이터 로드, 버튼 클릭 시 `axios.post` 또는 `fetch` 등).
- 요청과 응답 데이터 구조를 보고 어떤 정보를 주고받는지 파악하는 방법.
- API 호출 시 발생할 수 있는 다양한 오류 상황과, `.then / .catch` 또는 `try/catch`를 활용한 처리 방법.

마지막으로, Rolling API를 사용할 때 **항상 최신 문서**와 Swagger 정의를 참고하여 필드명이나 요청 형식이 정확한지 확인하는 습관을 가지세요. 이 가이드가 롤링페이퍼 서비스를 구현하는 데 도움이 되길 바랍니다.
