

# day50-rn-customhooks

태그	
날짜	@2022년 12월 9일

Hooks도 컴포넌트처럼 자신만의 Hook을 만들 수 있다.

만들 Hook함수는 특정 API에 GET요청을 보내고 응답을 받는 함수이다.

리액트 네이티브에서는 네트워크 통신을 위해 Fetch와 XMLHttpRequest를 제공하고, 추가적으로 WebSocket도 지원한다. 이번에는 Fetch를 이용하여 useFetch라는 이름의 Hook을 만들겠다.

src폴더 밑에 hooks라는 폴더를 만들고 useFetch Hook 함수를 작성할 useFetch.js파일을 생성하자.

```
src > hooks > JS useFetch.js > ...
1  import { useState,useEffect } from "react";
2
3  export const useFetch = url => {
4      const [date, setDate] = useState(null);
5      const [error, setError] = useState(null);
6
7      useEffect(async ()=> {
8          try{
9              const res = await fetch(url);
10             const result = await res.json();
11             if(res.ok){
12                 setData(result);
13                 setError(null);
14             }else {
15                 throw result;
16             }
17         }catch (error) {
18             setError(error);
19         }
20     }, []);
21     return { data, error };
22 };
```

전달받은 API의 주소로 요청을 보내고 그 결과를 성공 여부에 따라 data 혹은 error에 담아서 반환하는 useFetch를 만들었다. 이제 만들어진 useFetch를 이용해 API 요청하는 Dog 컴포넌트를 만들자.

The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with a 'src' directory containing 'components' and 'hooks'. The 'components' directory contains 'Button.js', 'Counter.js', 'Dog.js', 'Form.js', and 'Length.js'. The 'hooks' directory contains 'useFetch.js'. The 'Dog.js' file is selected in the file explorer. The code editor shows the following code:

```
src > components > JS Dog.js > ...
1  import React from "react";
2  import styled from "styled-components/native";
3  import { useFetch } from "../hooks/useFetch";
4
5  const StyledImage = styled.Image`
6    background-color : #7f8c8d;
7    width : 300px;
8    height : 300px;
9  `;
10 const ErrorMessage = styled.Text`
11   font-size : 18px;
12   color : #e74c3c;
13 `;
14
15 const URL = 'https://dog.ceo/api/breeds/image/random';
16 const Dog = () => {
17   const {data, error} = useFetch(URL);
18
19   return(
20     <>
21       <StyledImage source={data?.message ? { uri : data.message } : null} />
22       <ErrorMessage>{error?.message}</ErrorMessage>
23     </>
24   );
25 };
26
27 export default Dog;
28
29
```

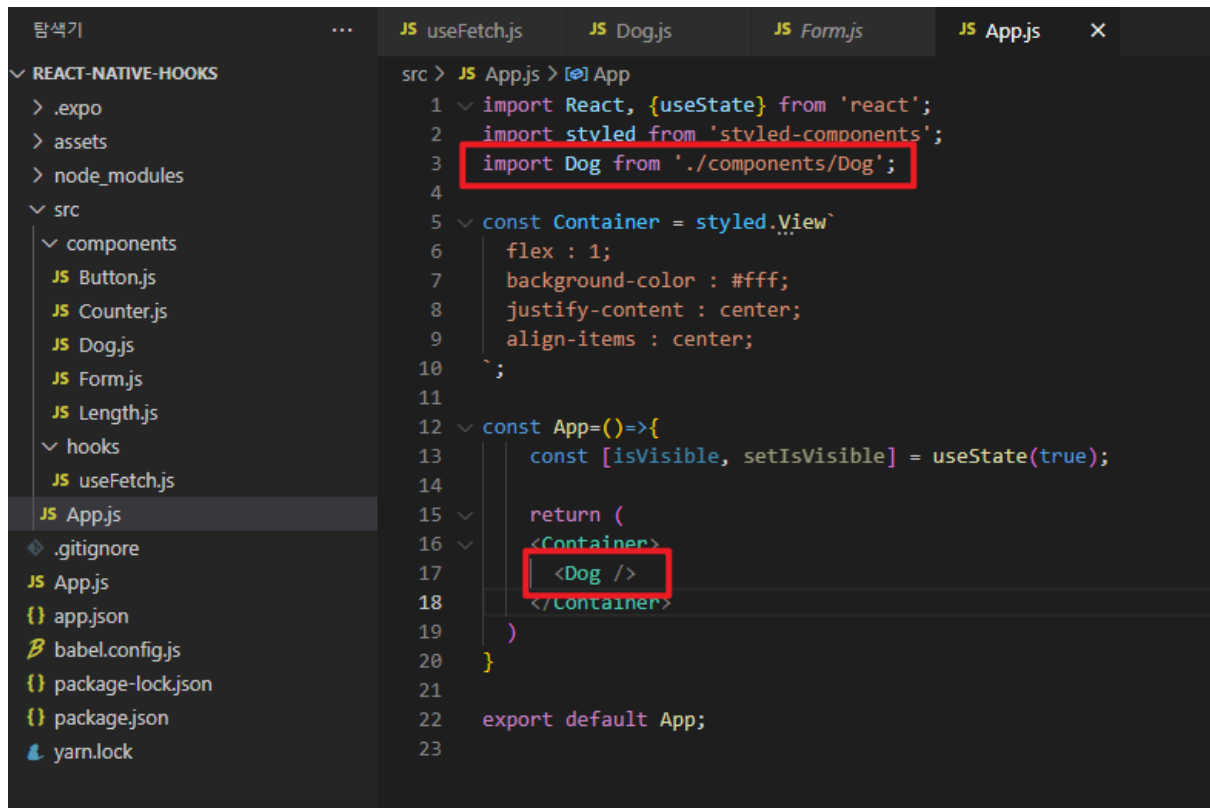
Dogs API를 이용해 무작위로 강아지 사진을 받아오는 컴포넌트를 만들었다.

- Dogs API : <https://dog.ceo/dog-api/>

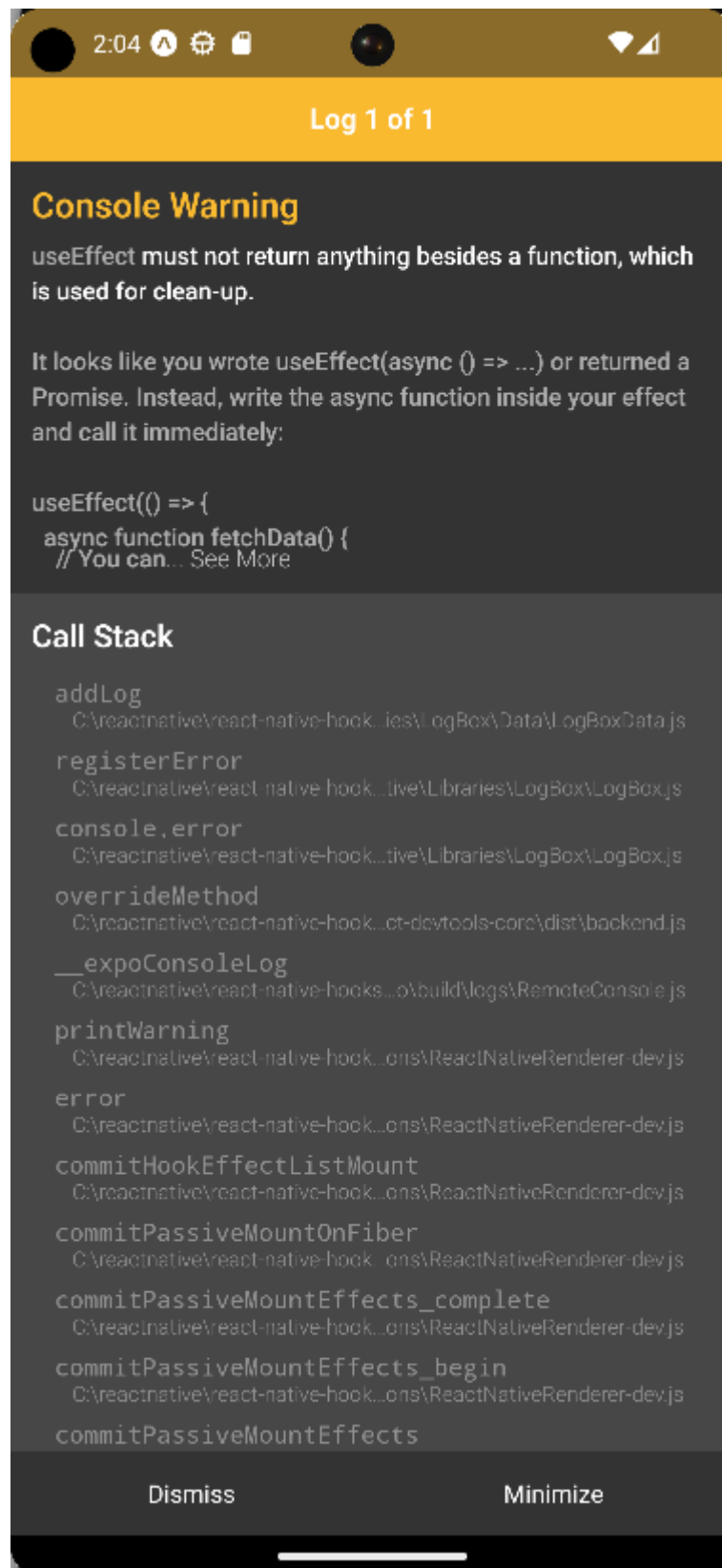
※ Dogs API 외에도 깃허브 public apis 레포지터리(repository)에서 다양한 API를 소개하고 있다.

- public-apis : <https://github.com/public-apis/public-apis>

이제 작성한 Dog 컴포넌트를 App컴포넌트에서 사용하고 결과를 확인해보자



```
src > JS App.js > App
1  import React, {useState} from 'react';
2  import styled from 'styled-components';
3  import Dog from './components/Dog';
4
5  const Container = styled.View`
6    flex : 1;
7    background-color : #fff;
8    justify-content : center;
9    align-items : center;
10 `;
11
12 const App=()=>{
13   const [isVisible, setIsVisible] = useState(true);
14
15   return (
16     <Container>
17       <Dog />
18     </Container>
19   )
20 }
21
22 export default App;
23
```



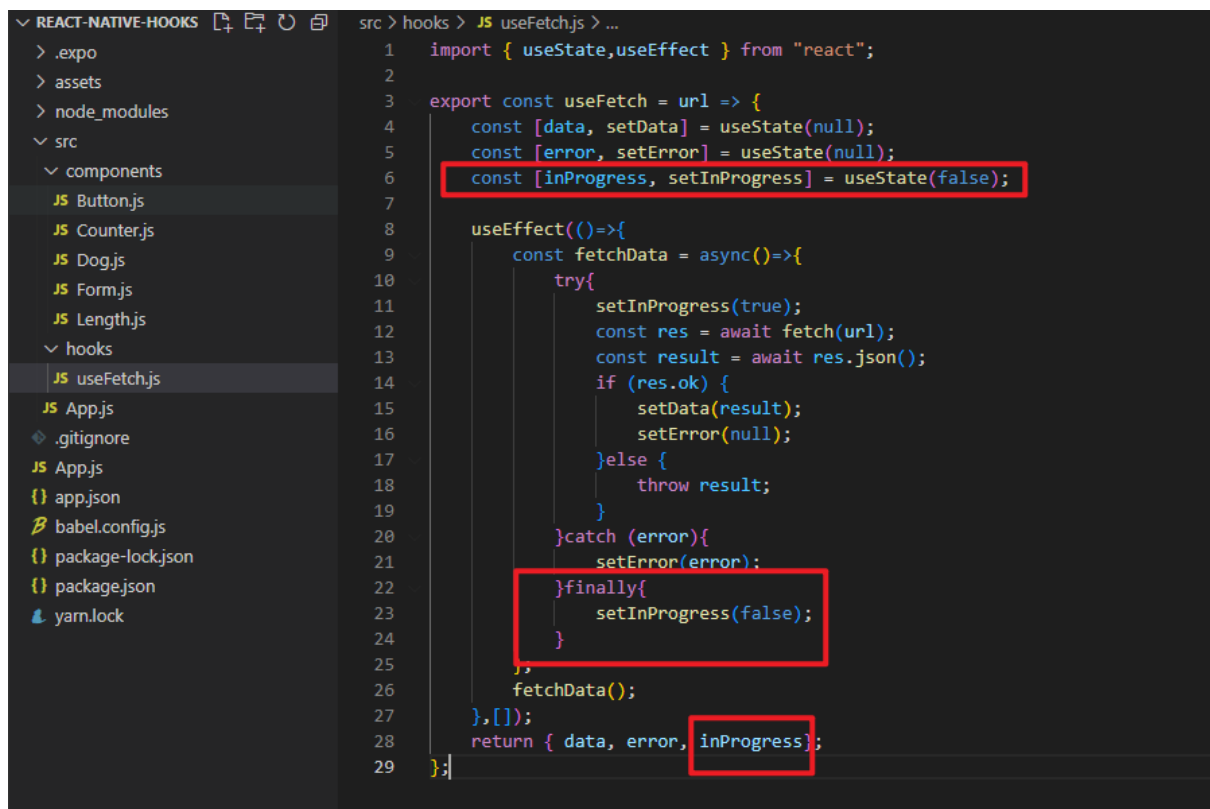
이런 경고 메시지가 나온다.

이 메시지는 `useEffect`의 첫 번째 파라미터로 비동기 함수를 전달했기 때문에 나타나는 경고 메시지이다. 비동기 함수를 이용해야 하는 상황에서는 `useEffect`에 전달되는 함수 내부

에 비동기 함수를 정의하고 사용하는 방법으로 이 문제를 해결할 수 있다.

에러가 나진 않지만 나는 사진이 나오지 않았다.

대부분의 비동기 작업 화면에서는 작업이 완료되기 전에 화면 전체 혹은 특정 버튼들이 사용할 수 없는 상태로 변경된다. API 요청을 보내는 비동기 동작에서는 선행된 작업이 마무리 되기 전에 추가적인 요청이 들어오지 않도록 화면을 구성하는 것이 좋다. 이를 위해 useFetch로부터 API 요청의 진행 상태를 알 수 있어야 비동기 요청의 작업 상태에 따라 화면 구성을 다르게 할 수 있다.



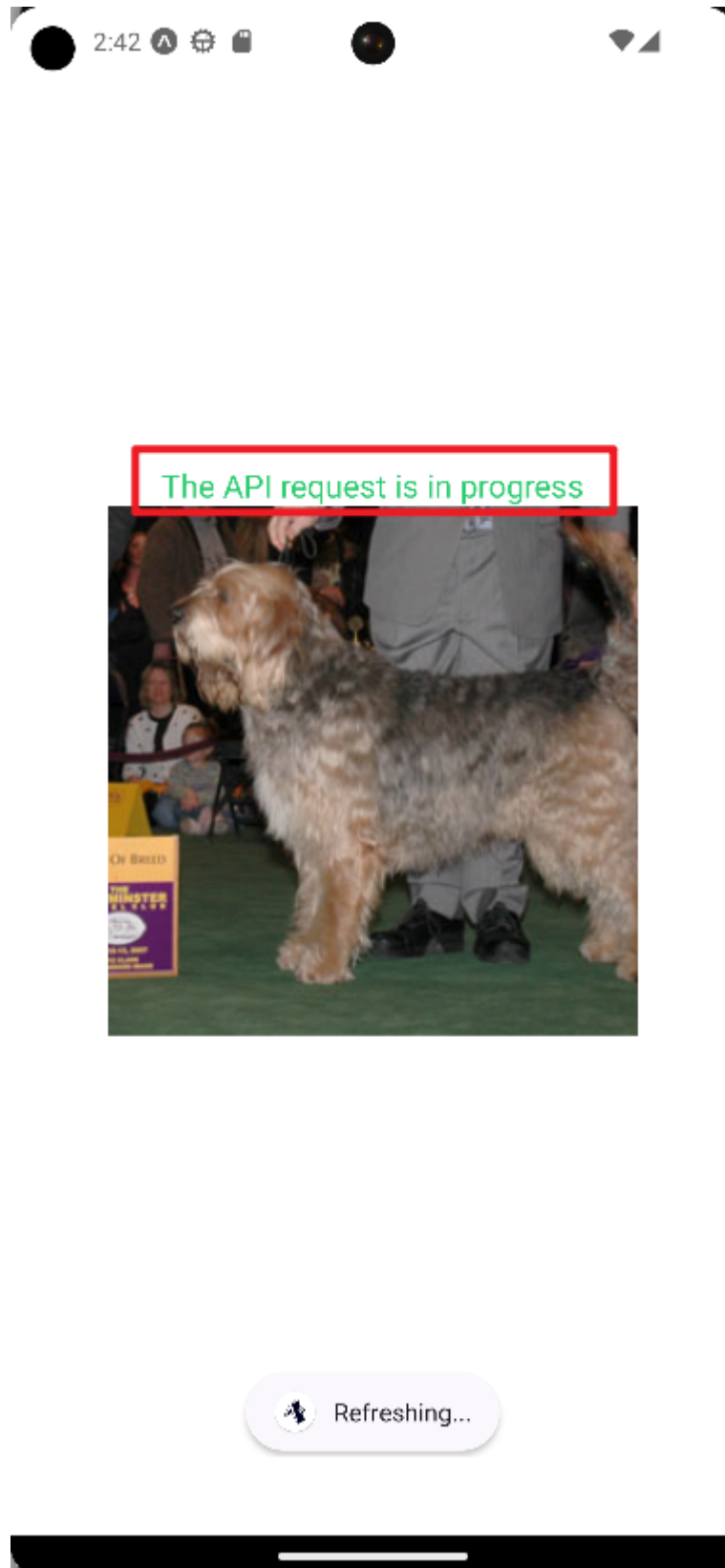
```
1  import { useState,useEffect } from "react";
2
3  export const useFetch = url => {
4    const [data, setData] = useState(null);
5    const [error, setError] = useState(null);
6    const [inProgress, setInProgress] = useState(false);
7
8    useEffect(()=>{
9      const fetchData = async()=>{
10        try{
11          setInProgress(true);
12          const res = await fetch(url);
13          const result = await res.json();
14          if (res.ok) {
15            setData(result);
16            setError(null);
17          }else {
18            throw result;
19          }
20        }catch (error){
21          setError(error);
22        }finally{
23          setInProgress(false);
24        }
25      }
26      fetchData();
27    },[]);
28    return { data, error, inProgress};
29  };
```

API의 진행 상태를 관리하는 inProgress를 만들고, API요청 시작 전과 완료 후 상태를 변경 해서 useFetch의 API 진행 상태를 확인할 수 있도록 수정했다.

이제 Dog컴포넌트에서 API 의 진행 상태를 확인하는 코드를 추가하자.

```
src > components > JS Dog.js > Dog
1  import React from "react";
2  import styled from "styled-components/native";
3  import { useFetch } from "../hooks/useFetch";
4
5  const StyledImage = styled.Image`
6    background-color : #7f8c8d;
7    width : 300px;
8    height : 300px;
9  `;
10 const ErrorMessage = styled.Text`
11   font-size : 18px;
12   color : #e74c3c;
13 `;
14 const LoadingMessage = styled.Text`
15   font-size : 18px;
16   color : #2ecc71;
17 `;
18
19 const URL = 'https://dog.ceo/api/breeds/image/random';
20 const Dog = () =>{
21   const {data, error, inProgress} = useFetch(URL);
22
23   return(
24     <>
25       {inProgress && (
26         <LoadingMessage>The API request is in progress</LoadingMessage>
27       )}
28       <StyledImage source={data?.message ? { uri: data.message } : null} />
29       <ErrorMessage>{error?.message}</ErrorMessage>
30     </>
31   );
32 };
33
34 export default Dog;
```

API요청 중에 진행되는 중에 LoadingMessage 컴포넌트가 잘 렌더링 된다.



위에 사진이 나오지 않았었는데, 찾아보니 오타가 있었다.

한글자 빠진것 만으로도 에러가 발생하고 어디인지 나오질 않으니 꼼꼼하게 코딩해야겠다.

hook도 컴포넌트와 마찬가지로 자주 사용하는 부분을 분리하면 코드가 깔끔해질 뿐만 아니라 여러 곳에서 재사용이 가능하다는 장점도 있다.