

day54-rn-chatapp3

☰ 태그	
📅 날짜	@2022년 12월 15일

메인화면

대부분의 애플리케이션에서 사용자들의 데이터 혹은 서비스의 데이터를 이용하려면 데이터에 접근할 수 있는 유효한 사용자라는 것을 증명해야 하므로 어떤 방법으로든 인증해야 한다. 인증 후에는 서비스를 이용할 수 있는 화면이 렌더링되고, 로그아웃 등으로 인증 상태를 해제하면 다시 인증을 위한 화면으로 이동한다.

지금까지 만든 AuthStack 네비게이션에서 사용되는 화면들은 인증 이전에 사용되는, 인증을 위한 화면이다. 화면들은 인증 후 사용되지 않다가 인증 상태가 해제되면 다시 렌더링되어야 한다.

인증 후에 사용될 화면과 화면들을 관리하는 네비게이션을 만들어보자.

채널과 관련된 3개의 화면과 사용자의 정보를 보여주는 프로필 화면이 있다. 채널과 관련된 화면은 스택 네비게이션을 이용하여 구성하고, 채널 관련 화면과 프로필 화면은 탭 네비게이션을 이용해 화면을 이동할 수 있도록 구성하자.

그림 p.382

네비게이션

먼저 인증에 성공하면 AuthStack 네비게이션 대신 렌더링할 MainStack 네비게이션의 화면들을 만들어 보자. MainStack 네비게이션은 채널 목록 화면과 프로필 화면으로 구성된 MainTab 네비게이션을 첫 번째 화면으로 가지며 그 외에 채널 생성 화면과 채널 화면으로 구성된다.

MainStack 네비게이션

MainStack 네비게이션을 구성하기 위해 채널 생성 화면과 채널 화면을 만들자.

```
JS images.js JS ChannelCreation.js X
src > screens > JS ChannelCreation.js > [🔍] default
1  import React from "react";
2  import styled from "styled-components";
3  import {Text, Button} from 'react-native'
4
5  const Container = styled.View`
6    flex : 1;
7    background-color : ${({theme}) => theme.background};
8  `;
9
10 const ChannelCreate = ({ navigation }) =>{
11   return (
12     <Container>
13       <Text style={{ fontSize : 24 }}>Channel Create</Text>
14       <Button title="Channel" onPress={() => navigation.navigate('Channel')} />
15     </Container>
16   );
17 };
18
19 export default ChannelCreate;
```

채널 화면으로 이동할 수 있는 버튼을 가진 간단한 채널 생성 화면을 만들었다.
다음으로 아래 코드처럼 작성하여 채널 화면을 만들자.

```
JS index.js JS ChannelCreation.js X
src > screens > JS ChannelCreation.js > [🔍] ChannelCreation
1  import React from "react";
2  import styled from "styled-components";
3  import {Text, Button} from 'react-native'
4
5  const Container = styled.View`
6    flex : 1;
7    background-color : ${({theme}) => theme.background};
8  `;
9
10 const ChannelCreation = ({ navigation }) =>{
11   return (
12     <Container>
13       <Text style={{ fontSize : 24 }}>Channel Create</Text>
14       <Button title="Channel" onPress={() => navigation.navigate('Channel')} />
15     </Container>
16   );
17 };
18
19 export default ChannelCreation;
```

화면이 작성이 모두 완료되면 screen 폴더의 index.js 파일을 아래코드처럼 수정하자.

```
JS index.js X JS ChannelCreation.js
src > screens > JS index.js
1  import Login from './Login';
2  import Signup from './Signup';
3  import Channel from './Channel';
4  import ChannelCreation from './ChannelCreation';
5
6  export {Login, Signup, Channel, ChannelCreation};
```

이제 navigations 폴더 밑에 MainStack.js 파일을 생성하고 준비된 화면을 이용해서 MainStack 내비게이션을 작성해보자.

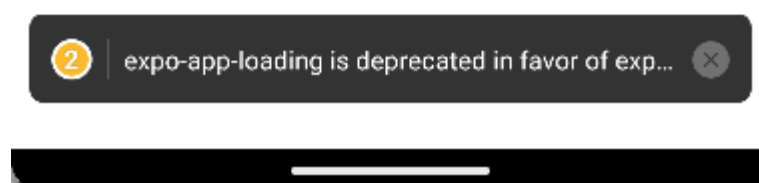
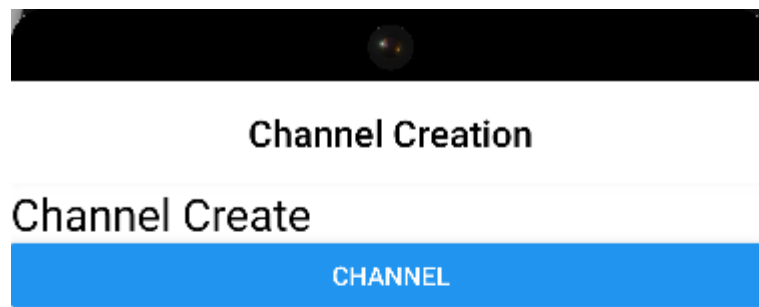
```
JS index.js JS MainStack.js X JS ChannelCreation.js
src > navigations > JS MainStack.js > [0] default
1  import React, {useContext} from "react";
2  import { ThemeContext } from "styled-components";
3  import { createStackNavigator } from "@react-navigation/stack";
4  import { Channel, ChannelCreation } from '../screens';
5
6  const Stack = createStackNavigator();
7  const MainStack = ()=>{
8    const theme = useContext(ThemeContext);
9
10    return (
11      <Stack.Navigator
12        screenOptions={{
13          headerTitleAlign : 'center',
14          headerTintColor : theme.headerTintColor,
15          cardStyle : {backgroundColor : theme.backgroundColor},
16          headerBackTitleVisible : false,
17        }}
18      >
19        <Stack.Screen name="Channel Creation" component={ChannelCreation} />
20        <Stack.Screen name="Channel" component={Channel} />
21      </Stack.Navigator>
22    );
23  };
24
25  export default MainStack;
26
```

헤더의 타이틀은 중앙으로 정렬하고, 버튼의 타이틀은 렌더링되지 않도록 설정했다.

headerTintColor 는 AuthStack 네비게이션과 동일하게 설정했다. 이제 navigations 폴더의 index.js 파일을 수정해서 MainStack 네비게이션이 잘 동작하는지 확인하자.

```
JS index.js ...\screens JS MainStack.js JS index.js ...\navigations X JS ChannelCreation.js
src > navigations > JS index.js > ...
1  import React, {useContext} from 'react';
2  import { NavigationContainer } from '@react-navigation/native';
3  import Spinner from '../components/Spinner'
4  import { ProgressContext } from '../contexts';
5  import MainStack from './MainStack';
6
7  const Naviagation = ()=>{
8      const {inProgress} = useContext(ProgressContext);
9      return(
10         <NavigationContainer>
11             <MainStack />
12             {inProgress && <Spinner />}
13         </NavigationContainer>
14     )
15 }
16
17 export default Naviagation;
```

MainStack 네비게이션의 동작을 확인하기 위해 AuthStack 네비게이션 대신 렌더링되도록 index.js파일을 수정했다.





MainTab 네비게이션

일반적으로 Screen 컴포넌트에는 화면으로 사용될 컴포넌트를 지정하지만, 내비게이션도 결국 컴포넌트이기 때문에 화면으로 사용할 수 있다.

MainStack 내비게이션에서 화면으로 사용되는 MainTab 내비게이션을 만들어보자.
MainTab 내비게이션을 구성하는 채널 목록 화면과 프로필 화면을 작성하자.

```
JS ChannelList.js X
src > screens > JS ChannelList.js > [0] default
1  import React from "react";
2  import styled from "styled-components";
3  import {Text, Button} from 'react-native';
4
5  const Container = styled.View`
6    flex : 1;
7    background-color : ${({theme}) => theme.background};
8  `;
9
10 const Channellist = ({navigation}) =>{
11   return(
12     <Container>
13       <Text style={{ fontSize : 24}} > Channel List</Text>
14       <Button
15         title="Channel Creation"
16         onPress={()=> navigation.navigate('Channel Creation')}
17       />
18     </Container>
19   );
20 };
21 export default Channellist;
```

화면을 확인할 수 있는 문자열과 채널 생성 화면으로 이동하는 버튼을 가진 채널 목록 화면을 만들었다. 채널 목록 화면과 함께 MainTab 내비게이션의 화면으로 사용된 프로필 화면을 만들자.

```
JS ChannelList.js JS Profile.js X
src > screens > JS Profile.js > Profile
1  import React from "react";
2  import styled from "styled-components";
3  import { Text } from "react-native";
4
5  const Container = styled.View`
6    flex : 1;
7    background-color : ${({theme})=> theme.background};
8  `;
9
10 const Profile = ()=>{
11   return(
12     <Container>
13       <Text style={{fontSize : 24}}>Profile</Text>
14     </Container>
15   );
16 };
17
18 export default Profile;
```

프로필 화면 작성까지 완료되면 screens 폴더의 index.js 파일을 아래 코드처럼 수정하자.

```
JS ChannelList.js JS Profile.js JS index.js X
src > screens > JS index.js
1  import Login from './Login';
2  import Signup from './Signup';
3  import Channel from './Channel';
4  import ChannelCreation from './ChannelCreation';
5  import Channellist from './Channellist';
6  import Profile from './Profile';
7
8  export {[Login,Signup, Channel, ChannelCreation, Channellist, Profile]};
```

navigations 폴더 밑에 MainTab.js파일을 생성하고 앞에서 작성한 두 화면을 이용해 아래 코드처럼 작성하자.


```

JS MainTab.js X
src > navigations > JS MainTab.js > [🔍] default
1  import React from "react";
2  import { createBottomTabNavigator } from "@react-navigation/bottom-tabs";
3  import { Profile, ChannelList } from "../screens";
4
5  const Tab = createBottomTabNavigator();
6
7  const MainTab = () => {
8    return(
9      <Tab.Navigator>
10       <Tab.Screen name="Channel List" component={ChannelList} />
11       <Tab.Screen name="Profile" component={Profile} />
12     </Tab.Navigator>
13   );
14 };
15 export default MainTab;

```

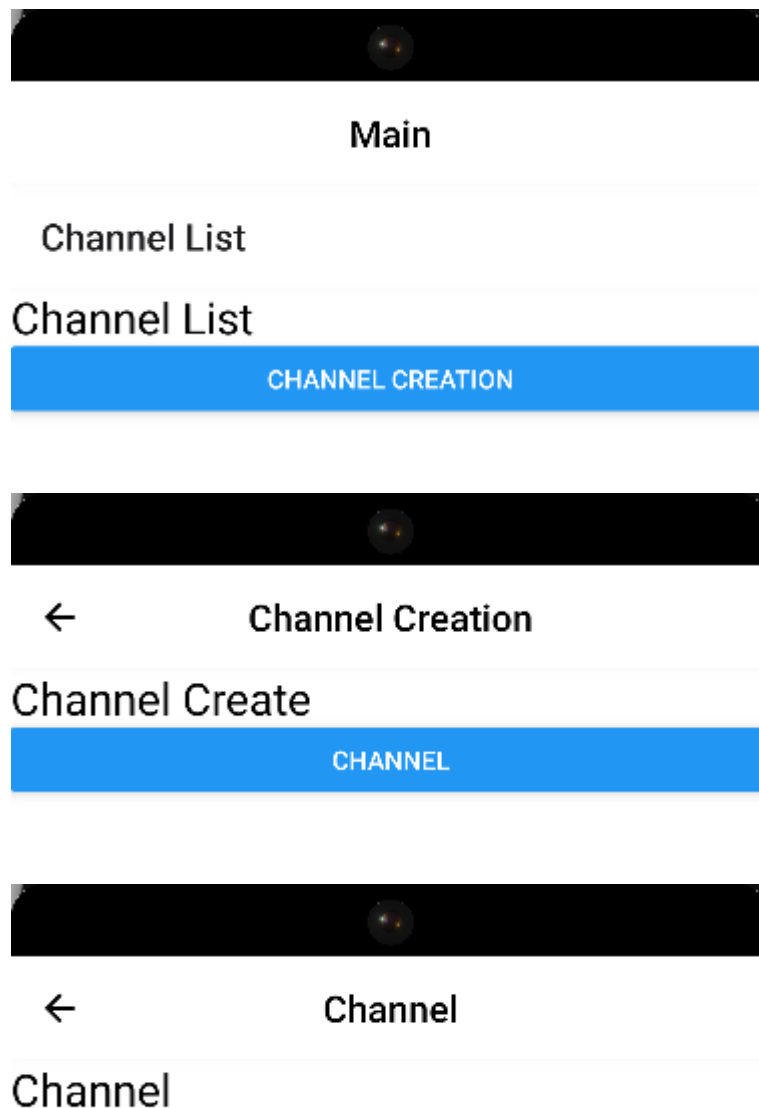
MainTab 내비게이션을 이용해서 MainStack 내비게이션을 아래 코드처럼 수정하자.

```

JS MainTab.js JS MainStack.js X
src > navigations > JS MainStack.js > [🔍] MainStack
1  import React, {useContext} from "react";
2  import { ThemeContext } from "styled-components";
3  import { createStackNavigator } from "@react-navigation/stack";
4  import { Channel, ChannelCreation } from '../screens';
5  import MainTab from "../MainTab";
6
7  const Stack = createStackNavigator();
8  const MainStack = () =>{
9    const theme = useContext(ThemeContext);
10
11    return (
12      <Stack.Navigator
13        initialRouteName="Main"
14        screenOptions={{
15          headerTitleAlign : 'center',
16          headerTintColor : theme.headerTintColor,
17          cardStyle : {backgroundColor : theme.backgroundColor},
18          headerBackTitleVisible : false,
19        }}
20      >
21        <Stack.Screen name="Main" component={MainTab} />
22        <Stack.Screen name="Channel Creation" component={ChannelCreation} />
23        <Stack.Screen name="Channel" component={Channel} />
24      </Stack.Navigator>
25    );
26  };
27
28  export default MainStack;
29

```

MainTab 내비게이션을 MainStack 내비게이션의 첫 번째 화면으로 렌더링되도록 수정했다.



인증과 화면 전환

인증상태에 따라 MainStack 내비게이션과 AuthStack 내비게이션을 렌더링하는 방법에 대해 알아보자. 애플리케이션이 시작되면 AuthStack 내비게이션이 렌더링되고, 로그인 혹은 회원가입을 통해 인증을 성공하면 MainStack 내비게이션이 렌더링 되어야 한다. 인증 후 로그아웃을 통해 인증 상태가 사라지면 다시 AuthStack 내비게이션이 렌더링 되어야 한다.

여러 곳에서 상태를 변경해야 할 경우 어떻게 관리하는 것이 좋을까. 앞에서 Spinner 컴포넌트의 렌더링 상태를 관리했던 것처럼 Context API를 이용하면 수월하게 전역적으로 관리할 수 있다. UserContext를 만들고 인증 상태에 따라 적절한 내비게이션이 렌더링 되도록 만들자.

```
JS User.js  X
src > components > JS User.js > ...
1  import React, {useState, createContext} from "react";
2
3  const UserContext = createContext({
4    user : { emai: null, uid : null},
5    dispatch : ()=>{},
6  })
7
8  const UserProvider = ({ children }) => {
9    const [user, setUser] = useState({});
10   const dispatch = ({email, uid }) => {
11     setUser({ email, uid });
12   };
13   const value = {user, dispatch};
14   return <UserContext.Provider value={value}>{children}</UserContext.Provider>;
15 };
16
17 export {UserContext, UserProvider};
```

UserContext 를 만들고 사용자의 이메일과 uid를 가진 user객체와 user 객체를 수정할 수 있는 dispatch 함수를 value로 전달하는 UserProvider 컴포넌트를 만들었다. 이제 contexts폴더의 index.js파일을 아래 코드처럼 수정하자.

```
JS User.js  JS index.js  X
src > contexts > JS index.js
1  import { ProgressContext, ProgressProvider } from "../Progress";
2  import { UserContext, UserProvider } from "../components/User";
3
4  export { ProgressContext, ProgressProvider, UserContext, UserProvider };
```

이제 앞에서 만든 UserProvider 컴포넌트를 이용해서 App 컴포넌트에 작성된 컴포넌트를 모두 감쌀 수 있도록 수정하자.

```
JS User.js JS index.js JS App.js X
src > JS App.js > ...
1
2 import React, { useState } from 'react';
3 import { StatusBar, Image } from 'react-native';
4 import AppLoading from 'expo-app-loading';
5 import { Asset } from 'expo-asset';
6 import * as Font from 'expo-font';
7 import { ThemeProvider } from 'styled-components';
8 import { theme } from './theme';
9 import Navigation from './navigations';
10 import { images } from './utils/images';
11 import { ProgressProvider, UserProvider } from './contexts';
12
```

```
return isReady ?(
  <ThemeProvider theme={theme}>
    <UserProvider>
      <ProgressProvider>
        <StatusBar barStyle="dark-content" />
        <Navigation />
      </ProgressProvider>
    </UserProvider>
  </ThemeProvider>
)
```

이제 UserContext의 user 상태에 따라 인증 여부를 확인할 수 있다. 인증 여부에 따라 MainStack 네비게이션 혹은 AuthStack 네비게이션이 렌더링되도록 navigations 폴더의 index.js 파일을 수정하자.

```
JS index.js X JS App.js JS User.js JS Login.js JS Signup.js
src > navigations > JS index.js > [⌘] default
1  import React, {useContext} from 'react';
2  import { NavigationContainer } from '@react-navigation/native';
3  import Spinner from '../components/Spinner';
4  import { ProgressContext, UserContext } from '../contexts';
5  import MainStack from './MainStack';
6  import AuthStack from './AuthStack';
7
8  const Navigation = () => {
9    const {inProgress} = useContext(ProgressContext);
10   const {user} = useContext(UserContext);
11   return(
12     <NavigationContainer>
13       {user?.uid && user?.email ? <MainStack /> : <AuthStack />}
14       {inProgress && <Spinner />}
15     </NavigationContainer>
16   )
17 }
18
19 export default Navigation;
```

UserContext의 user에 uid와 email 값이 존재하면 인증된 것으로 판단하고 MainStack 내 비게이션을 렌더링하도록 작성했다. 이제 인증되면 UserContext의 user를 수정하도록 로그인 화면을 수정하자.

```
JS index.js JS Login.js X
src > screens > JS Login.js > [⌘] Container
1  import React, {useState, useRef, useEffect, useContext} from 'react';
2  import { ProgressContext, UserContext } from '../contexts';
3  import styled from 'styled-components';
4  import { Image, Input, Button } from '../components';
5  import { images } from '../utils/images';
6  import { TouchableWithoutFeedback, Keyboard } from 'react-native';
7  import { KeyboardAwareScrollView } from 'react-native-keyboard-aware-scroll-view';
8  import { validateEmail, removeWhitespace } from '../utils/common';
9  import { useSafeAreaInsets } from 'react-native-safe-area-context';
10 import { Alert } from 'react-native';
11 import { signin } from '../utils/firebase';
```

```
const login = ({navigation})=>{
  const {dispatch} = useContext(UserContext);
  const [email,setEmail] = useState('');
  const [password, setPassword] = useState('');
  const passwordRef = useRef();
  //에러변수 담기
  const [errorMessage, setErrorMessage] = useState('');
  const insets = useSafeAreaInsets();
  const [disabled,setDisabled] = useState(true);
  const {spinner} = useContext(ProgressContext);
```

```
const _handleLoginButtonPress = async () => {
  try {
    spinner.start();
    const user = await signin({ email, password });
    dispatch(user);
    Alert.alert('Login Success',user.email);
  } catch (e) {
    Alert.alert('Login Error', e.message);
  } finally {
    spinner.stop;
  }
};
```

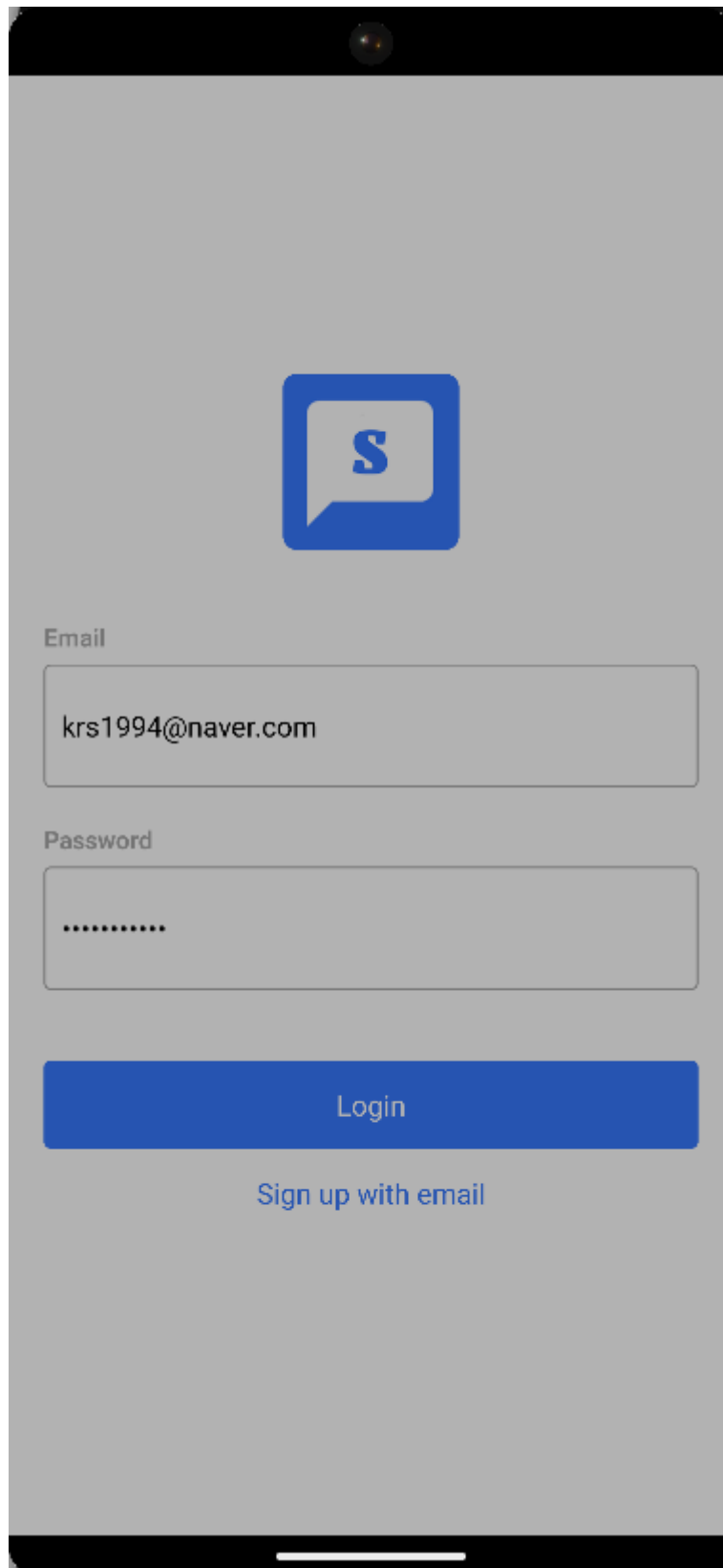
로그인에 성공하면 UserContext의 함수를 이용해 user의 상태가 인증된 사용자 정보로 변경되도록 작성했다. 이제 회원가입 화면에서도 회원가입 성공 시 UserContext의 user상태가 변경되도록 수정하자.

```
JS index.js JS Login.js JS Signup.js X
src > screens > JS Signup.js > [⌕] Signup > [⌕] _handleSignupButtonPress
1  import React, {useState, useRef, useEffect, useContext} from 'react';
2  import styled from 'styled-components';
3  import {Image, Input, Button} from '../components'
4  import { KeyboardAwareScrollView } from 'react-native-keyboard-aware-scroll-view';
5  import { validateEmail, removeWhitespace } from '../utils/common';
6  import {images} from '../utils/images'
7  import { Alert } from 'react-native';
8  import { signup } from '../utils/firebase';
9  import { ProgressContext, UserContext } from '../contexts';
10
```

```
const Signup = () => {
  const {dispatch} = useContext(UserContext);
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [passwordConfirm, setPasswordConfirm] = useState('');
  const [errorMessage, setErrorMessage] = useState('');
  const [disabled, setDisabled] = useState(true);
  const [photoUrl, setPhotoUrl] = useState(images.photo);
  const {spinner} = useContext(ProgressContext);
```

```
const _handleSignupButtonPress = async () => {
  try {
    spinner.start();
    const user = await signup ({email, password, name, photoUrl});
    dispatch(user);
    console.log(user);
    Alert.alert('Signup Success', user.email);
  } catch (e) {
    Alert.alert('Signup Error', e.message);
  } finally {
    spinner.stop();
  }
};
```

회원가입 화면도 로그인 화면과 마찬가지로 사용자 생성에 성공했을 때 반환되는 사용자 정보를 이용해서 UserContext의 user 상태가 변경되도록 dispatch 함수를 호출했다. 결과를 보면 인증 상태에 따라 렌더링되는 내비게이션이 변경되는 것을 확인할 수 있다.



A mobile application login screen with a light gray background. At the top center is a blue square icon containing a white speech bubble with a blue 'S' inside. Below the icon are two input fields: 'Email' containing 'krs1994@naver.com' and 'Password' containing eight dots. A blue 'Login' button is positioned below the password field. At the bottom, the text 'Sign up with email' is displayed in a blue, sans-serif font. The screen is framed by a black border with a small circular notch at the top center and a horizontal white line at the bottom center.

Email

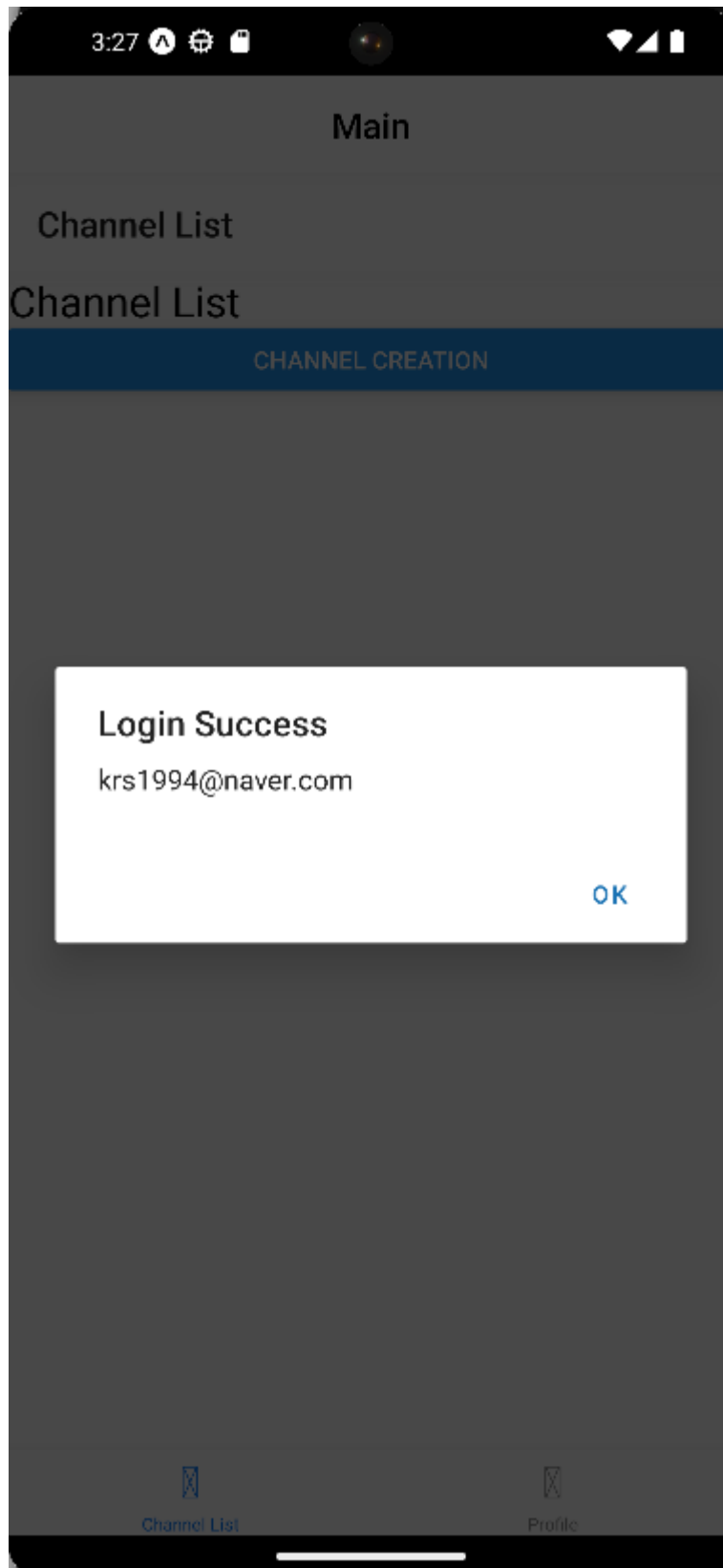
krs1994@naver.com

Password

.....

Login

Sign up with email



인증에 성공한 후 뒤로 가기 버튼을 통해 로그인 화면으로 돌아간다면 어색할 것이다.

이렇게 렌더링되는 내비게이션을 변경하면 스와이프 혹은 기기에 있는 뒤로 가기 버튼을 클릭해도 이전 내비게이션으로 돌아가지 않는다.

로그아웃을 통해 인증 상태를 해제하고 하디 AuthStack 내비게이션이 렌더링되도록 만들어 보자. 먼저 firebase.js 파일에 로그아웃 함수를 아래 코드처럼 만들자.

```
JS firebase.js X
src > utils > JS firebase.js > ...
22     return user;
23   };
24
25   export const signout = async () => {
26     await signOut(auth);
27     return {};
28   };
29
```

프로필 화면에서 Button 컴포넌트를 이용해 로그아웃 버튼을 생성하고 앞에서 작성한 logout함수를 호출하도록 작성했다.

```
JS firebase.js  JS Profile.js  X
src > screens > JS Profile.js > ...
1  import React, {useContext} from "react";
2  import styled from "styled-components";
3  import { Button } from "react-native";
4  import {logout} from '../utils/firebase';
5  import {UserContext} from '../contexts';
6
7  const Container = styled.View`
8    flex : 1;
9    background-color :${({theme})=> theme.background};
10 `;
11
12 const Profile =()=>{
13   const {dispatch} = useContext(UserContext);
14
15   const _handleLogoutButtonPress = async ()=>{
16     try {
17       await logout();
18     }catch(e) {
19       console.log('[Profile] logout : ', e.message);
20     }finally {
21       dispatch({});
22     }
23   };
24   return(
25     <Container>
26       <Button title="logout" onPress={_handleLogoutButtonPress} />
27     </Container>
28   );
29 };
30
31 export default Profile;
```

프로필 화면에 로그아웃 버튼을 추가하고 firebase.js 파일에 작성된 logout 함수가 호출되도록 작성했다. logout 함수가 완료되면 UserContext의 dispatch 함수를 이용해 user의 상태를 변경하고 AuthStack 내비게이션이 렌더링되도록 만들었다. 사용자 인증을 했을때와 마찬가지로 로그아웃을 통해 인증을 해제한 후에는 스와이프나 뒤로 가기 버튼을 통해 다시 이전 내비게이션 화면으로 돌아갈 수 없다.

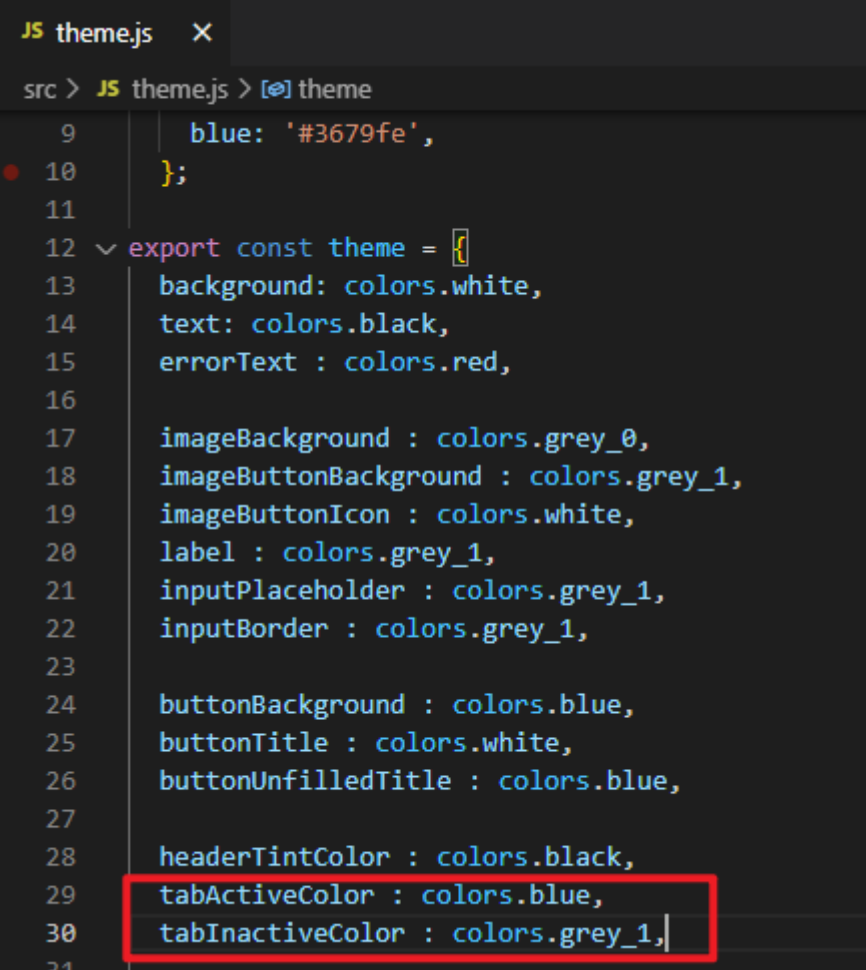


프로필 화면

MainTab 네비게이션의 탭 버튼을 수정하고 프로필 화면을 만들어보자. 탭버튼을 아이콘을 추가하고, 프로필 화면에서는 사용자의 사진을 변경할 수 있는 기능을 추가해 디자인을 변경해보자.

탭버튼 변경

탭 버튼에서 사용할 색을 theme.js 파일에 정의하고 진행하자.



```
JS theme.js  X
src > JS theme.js > [E] theme
9      blue: '#3679fe',
10     };
11
12  export const theme = {
13    background: colors.white,
14    text: colors.black,
15    errorText : colors.red,
16
17    imageBackground : colors.grey_0,
18    imageButtonBackground : colors.grey_1,
19    imageButtonIcon : colors.white,
20    label : colors.grey_1,
21    inputPlaceholder : colors.grey_1,
22    inputBorder : colors.grey_1,
23
24    buttonBackground : colors.blue,
25    buttonTitle : colors.white,
26    buttonUnfilledTitle : colors.blue,
27
28    headerTintColor : colors.black,
29    tabActiveColor : colors.blue,
30    tabInactiveColor : colors.grey_1,
31  }
```

MainTab 네비게이션을 수정해서 탭 버튼에 아이콘을 추가하고 활성화 상태에서 사용하는 색을 변경하자.

```
JS theme.js JS MainTab.js X
src > navigations > JS MainTab.js > [🔍] MainTab > 📁 tabBarIcon
1  import React, {useContext} from "react";
2  import { createBottomTabNavigator } from "@react-navigation/bottom-tabs";
3  import { Profile, Channellist } from "../screens";
4  import {MaterialIcons} from '@expo/vector-icons';
5  import { ThemeContext } from "styled-components";
6
7  const Tab = createBottomTabNavigator();
8
9  const TabBarIcon = ({focused, name}) => {
10    const theme = useContext(ThemeContext);
11    return(
12      <MaterialIcons
13        name={name}
14        size={26}
15        color={focused ? theme.tabActiveColor : theme.tabInactiveColor}
16      />
17    );
18  };
19
```

```

✓ const MainTab = () => {
  ✓ const theme = useContext(ThemeContext);
  ✓ return(
    ✓ <Tab.Navigator
      ✓ tabBarOptions={{
        ✓   activTintColor : theme.tabActiveColor,
        ✓   inactiveTintColor : theme.tabInactiveColor,
        ✓ }}>
      ✓ <Tab.Screen
        ✓   name="Channel List"
        ✓   component={Channellist}
        ✓   options={{
          ✓     tabBarIcon : ({focused}) =>
            ✓       TabBarIcon ({
              ✓         focused,
              ✓         name : focused ? 'chat-bubble' : 'chat-bubble-outline',
              ✓       }),
          ✓   }}/>
        ✓ <Tab.Screen name="Profile" component={Profile}
          ✓   options={{
            ✓     tabBarIcon : ({focused}) =>
              ✓       TabBarIcon ({
                ✓         focused,
                ✓         name : focused ? 'person' : 'person-outline',
                ✓       }),
            ✓   }}/>
        ✓ </Tab.Screen>
      ✓ </Tab.Navigator>
    ✓ );
  ✓ };
  export default MainTab;

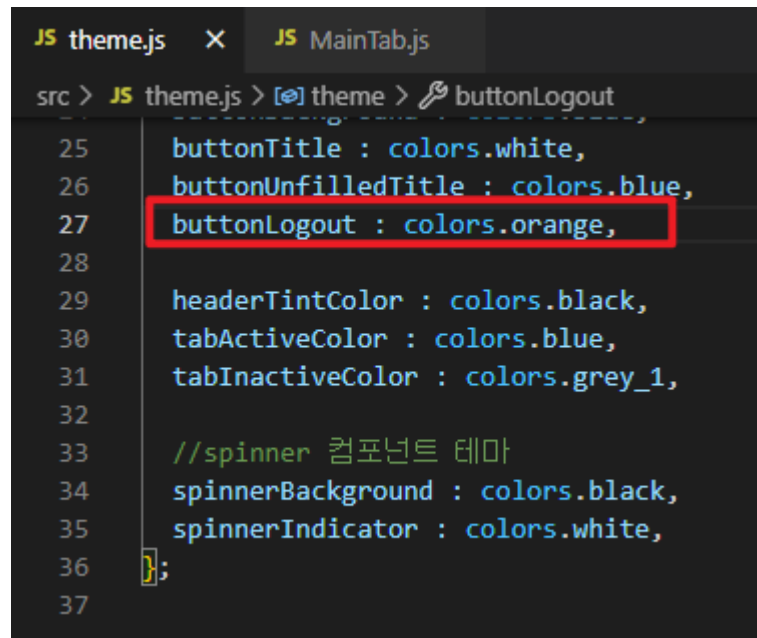
```



프로필 화면

프로필 화면은 사용자의 정보를 확인할 수 있고 사진을 변경할 수 있는 기능과 로그아웃 기능을 제공한다.

프로필화면에 있는 로그아웃 버튼의 배경색으로 사용할 값을 theme.js파일에 정의하자



```
25   buttonTitle : colors.white,
26   buttonUnfilledTitle : colors.blue,
27   buttonLogout : colors.orange,
28
29   headerTintColor : colors.black,
30   tabActiveColor : colors.blue,
31   tabInactiveColor : colors.grey_1,
32
33   //spinner 컴포넌트 테마
34   spinnerBackground : colors.black,
35   spinnerIndicator : colors.white,
36 };
37
```

현재 접속한 사용자의 정보를 반환하는 함수와 사용자의 사진을 수정하는 함수를 firebase.js 파일에 작성하자.

```
JS theme.js JS firebase.js X JS MainTab.js
src > utils > JS firebase.js > ...
1 import { initializeApp } from 'firebase/app';
2 import {
3   getAuth,
4   signInWithEmailAndPassword,
5   createUserWithEmailAndPassword,
6   signOut,
7   updateProfile,
8 } from 'firebase/auth';
9 import { getFirestore, collection, doc, setDoc } from 'firebase/firestore';
10 import { getDownloadURL, getStorage, ref, uploadBytes } from 'firebase/storage';
11 import config from '../../firebase.json';
12 export const app = initializeApp(config);
13 const auth = getAuth(app);
14 > export const signin = async ({ email, password }) => { ...
17 };
18 > export const signup = async ({ name, email, password, photoUrl }) => { ...
23 };
24
25 > export const signout = async () => { ...
28 };
29
30 > const uploadImage = async uri => { ...
43 };
44
45 > export const getCurrentUser = () => {
46   const { uid, displayName, email, photoURL } = auth.currentUser;
47   return { uid, name: displayName, email, photoUrl: photoURL };
48 };
49 > export const updateUserInfo = async photo => {
50   const photoUrl = await uploadImage(photo);
51   await updateProfile(auth.currentUser, { photoUrl });
52   return photoUrl;
53 };
```

현재 접속한 사용자의 정보가 있는 currentUser에서 필요한 값을 받아오는 함수를 작성하고, 스토리지에 선택된 사진을 업로드하는 함수를 이용해서 사용자의 사진을 수정하는 함수를 만들었다.

준비된 함수들을 이용해 프로필화면을 만들어 보자.

```

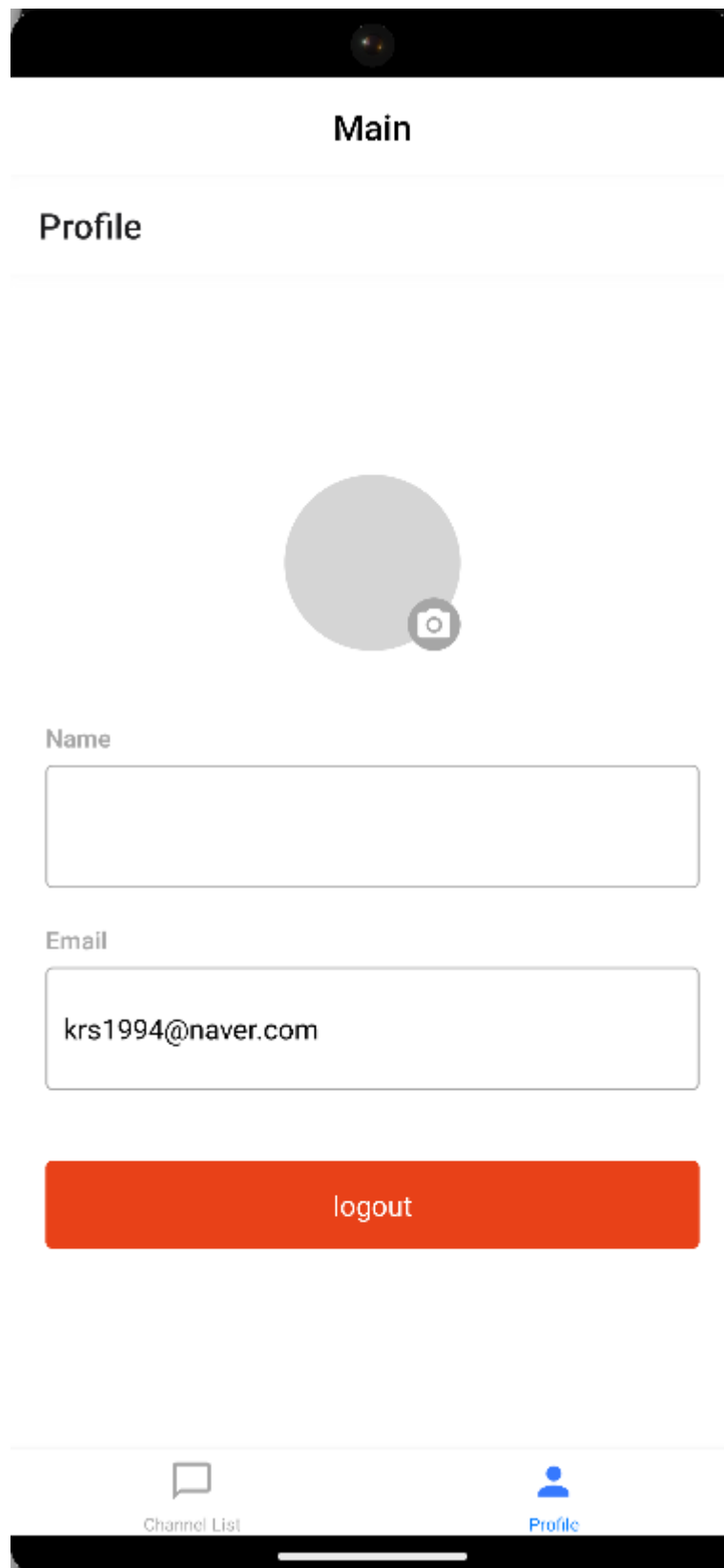
JS Profile.js  X  JS theme.js
src > screens > JS Profile.js > [0] Profile
1  import React,{useContext, useState} from "react";
2  import styled, { ThemeContext } from "styled-components";
3  import { Button ,Image, Input} from "../components";
4  import { UserContext, ProgressContext } from '../contexts';
5  import { getCurrentUser, updateUserInfo, signout } from '../utils/firebase';
6  import { Alert } from 'react-native';
7
8  const Container = styled.View`
9      flex: 1;
10     background-color: ${({ theme }) => theme.background};
11     justify-content: center;
12     align-items: center;
13     padding: 0 20px;
14 `;
15 const Profile = () => {
16     const {dispatch} = useContext(UserContext);
17     const { spinner } = useContext(ProgressContext);
18     const theme = useContext(ThemeContext);
19
20     const user = getCurrentUser();
21     const [photoUrl, setPhotoUrl] = useState(user.photoUrl);
22
23
24
25     const _handleLogoutButtonPress = async()=>{
26         try{
27             spinner.start();
28             await signout();
29         }catch(e){
30             console.log('[Profile] logout: ',e.message);
31         }finally{
32             dispatch({});
33             spinner.stop();
34         }
35     }

```

```

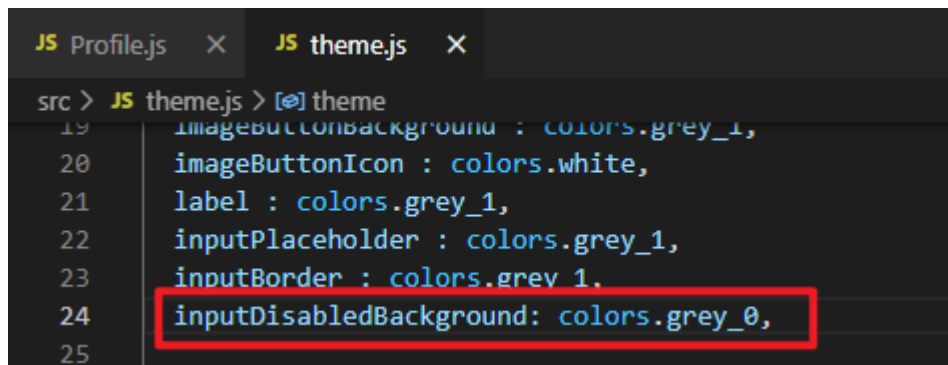
36
37   const _handlePhotoChange = async url => {
38     try {
39       spinner.start();
40       const photoUrl = await updateUserInfo(url);
41       setPhotoUrl(photoUrl);
42     } catch (e) {
43       Alert.alert('Photo Error', e.message);
44     } finally {
45       spinner.stop();
46     }
47   };
48
49
50
51   return(
52     <Container>
53     | <Image
54     url={photoUrl}
55     onChangeImage={_handlePhotoChange}
56     showButton
57     rounded
58     />
59
60     <Input label="Name" value={user.name} disabled />
61     <Input label="Email" value={user.email} disabled />
62     <Button
63     title="logout"
64     onPress={_handleLogoutButtonPress}
65     containerStyle={{ marginTop: 30, backgroundColor: theme.buttonLogout }}
66     />
67
68     </Container>
69   )
70 }
71
72 export default Profile;

```



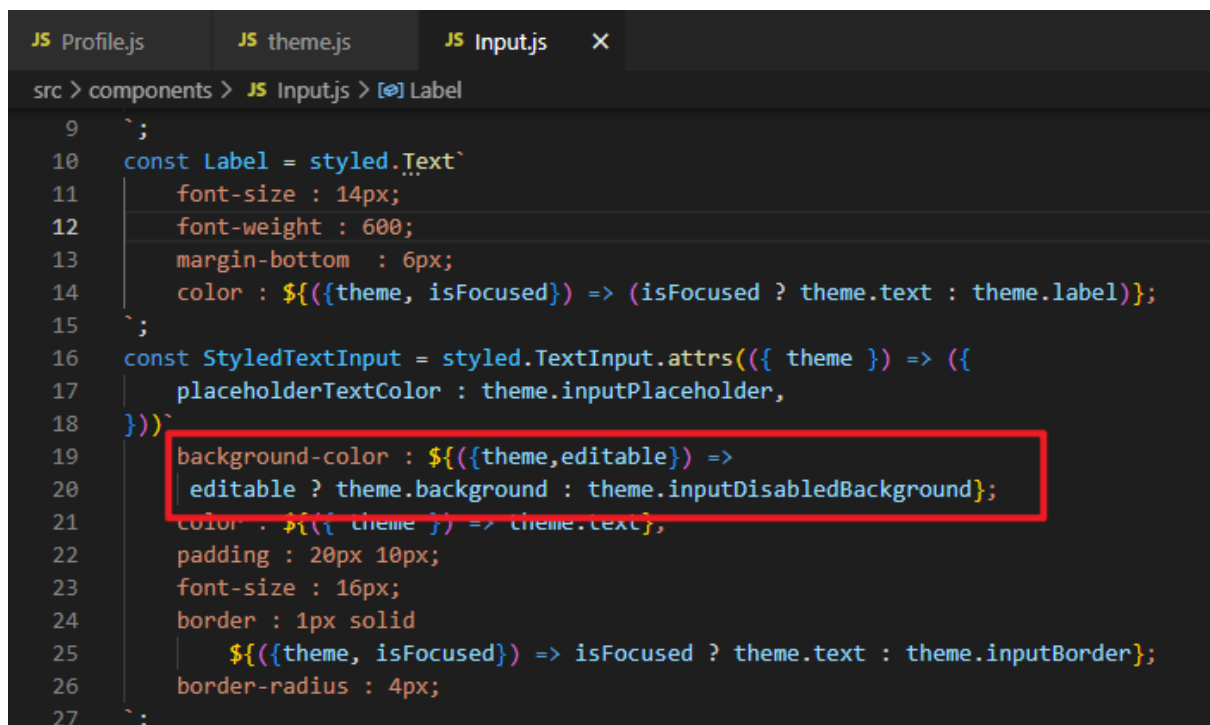
프로필 화면에서는 사용자의 이름이나 이메일을 수정하는 기능을 제공하지 않으므로 Input 컴포넌트에서 입력이 불가능하도록 수정되어야 한다. 추가적으로 Input 컴포넌트에서 반드시 전달되어야 하는 props가 전달되지 않아 경고 메시지가 나타나는 문제도 해결해야 한다.

비활성화된 Input 컴포넌트의 배경색을 theme.js로 정의하자.



```
JS Profile.js x JS theme.js x
src > JS theme.js > [0] theme
19 imageButtonBackground : colors.grey_1,
20 imageButtonIcon : colors.white,
21 label : colors.grey_1,
22 inputPlaceholder : colors.grey_1,
23 inputBorder : colors.grev 1,
24 inputDisabledBackground: colors.grey_0,
25
```

정의된 색을 이용해서 input 컴포넌트를 아래 코드처럼 수정하자.



```
JS Profile.js JS theme.js JS Input.js x
src > components > JS Input.js > [0] Label
9 `;
10 const Label = styled.Text`
11   font-size : 14px;
12   font-weight : 600;
13   margin-bottom : 6px;
14   color : ${({theme, isFocused}) => (isFocused ? theme.text : theme.label)};
15 `;
16 const StyledTextInput = styled.TextInput.attrs(({ theme }) => ({
17   placeholderTextColor : theme.inputPlaceholder,
18 })))`
19   background-color : ${({theme, editable}) =>
20     editable ? theme.background : theme.inputDisabledBackground};
21   color : ${({theme}) => theme.text},
22   padding : 20px 10px;
23   font-size : 16px;
24   border : 1px solid
25     ${({theme, isFocused}) => isFocused ? theme.text : theme.inputBorder};
26   border-radius : 4px;
27 `;
```

```
const Input = forwardRef(
  (
    {
      label,
      value,
      onChangeText,
      onSubmitEditing,
      onBlur,
      placeholder,
      isPassword,
      returnKeyType,
      maxLength,
      disabled,
    },
  ),
```

```
return(
  <Container>
    <Label isFocused={isFocused}>{label}</Label>
    <StyledTextInput
      ref={ref}
      isFocused={isFocused}
      value={value}
      onChangeText={onChangeText}
      onSubmitEditing={onSubmitEditing}
      onFocus={() => setIsFocused(true)}
      onBlur={() => {
        setIsFocused(false);
        onBlur();
      }}
      placeholder={placeholder}
      secureTextEntry={isPassword}
      returnKeyType={returnKeyType}
      maxLength={maxLength}
      autoCapitalize="none"
      autoComplete={false}
      keyboardType="none" // IOS ONLY
      underlineColorAndroid = "transparent" // Android only
      editable={!disabled}
    />
  </Container>
);
```

```

Input.defaultProps = {
  onBlur : () => {},
  onChangeText : () => {},
  onSubmitEditing : () => {},
};

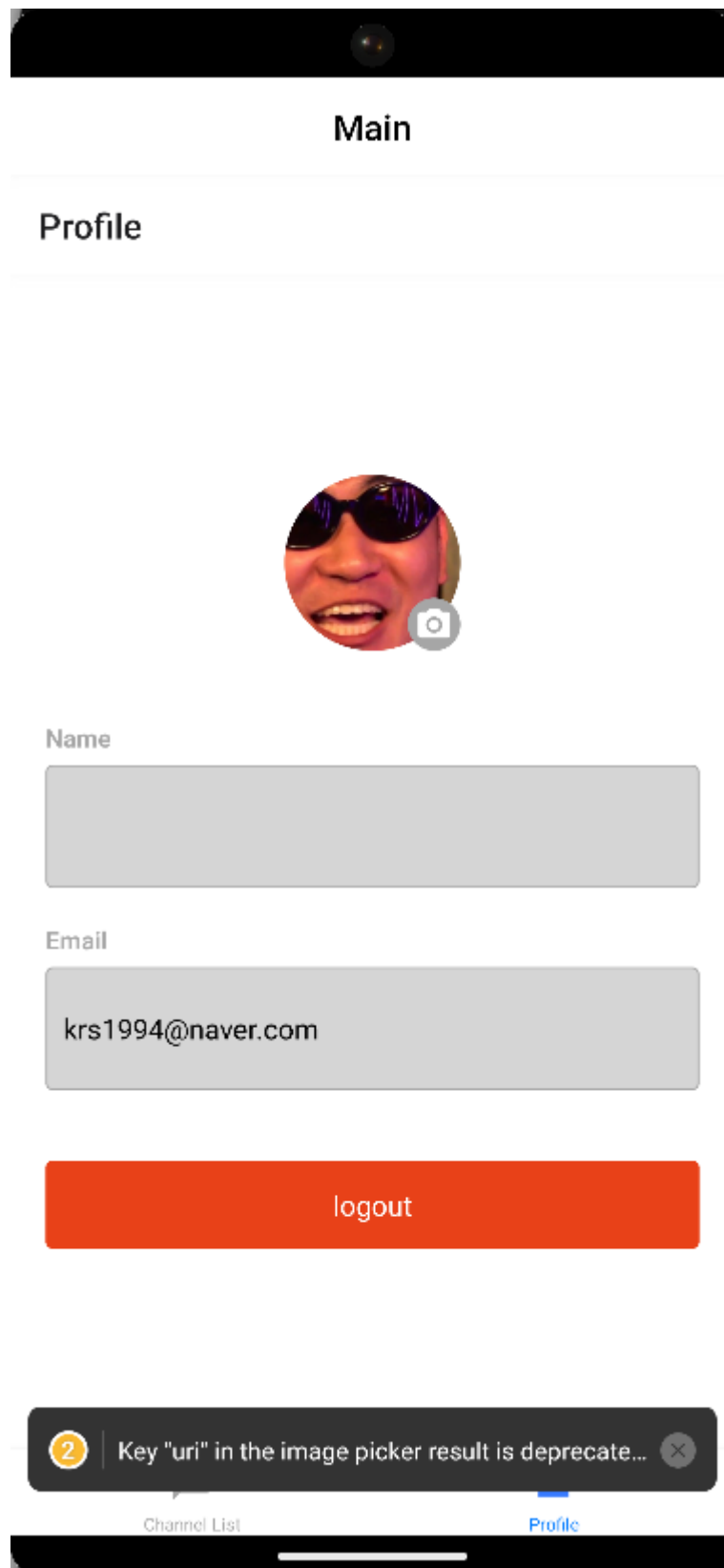
Input.propTypes = {
  label : PropTypes.string.isRequired,
  value : PropTypes.string.isRequired,
  onChangeText : PropTypes.func,
  onSubmitEditing : PropTypes.func,
  onBlur : PropTypes.func,
  placeholder : PropTypes.string,
  isPassword : PropTypes.bool,
  returnKeyType : PropTypes.oneOf(['done', 'next']),
  maxLength : PropTypes.number,
  disabled : PropTypes.bool,
};

export default Input;

```

propTypes에 필수로 지정되어 있던 onChangeText와 onSubmitEditing을 수정하고 사용 가능 여부를 결정하는 disabled를 추가했다. props로 전달되는 disabled를 이용해 리액트 네이티브의 TextInput 컴포넌트 활성화 여부를 결정하는 editable의 값이 변경되도록 하고, editable 값에 따라 배경색이 다르게 나타나도록 수정했다. 이제 프로필 화면에서 사용중인 input 컴포넌트를 수정할 수 없도록 하자.


```
JS Profile.js X JS theme.js JS Input.js
src > screens > JS Profile.js > Profile
51 return(
52   <Container>
53     <Image
54       url={photoUrl}
55       onChangeImage={_handlePhotoChange}
56       showButton
57       rounded
58     />
59
60     <Input label="Name" value={user.name} disabled />
61     <Input label="Email" value={user.email} disabled />
62     <Button
63       title="logout"
64       onPress={_handleLogoutButtonPress}
65       containerStyle={{ marginTop: 30, backgroundColor: theme.buttonLogout }}
66     />
67
68   </Container>
69 )
70 }
71
72 export default Profile;
```



헤더 변경

MainStack 내비게이션에서 MainTab 내비게이션이 화면으로 사용되는 Screen 컴포넌트의 name은 “Main”으로 설정되어 있다. 헤더의 타이틀과 관련해 특별히 설정하지 않으면 Screen 컴포넌트의 name에 설정된 값이 헤더의 타이틀로 되기 때문에, 프로필 화면과 채널 목록 화면 모두 “Main”으로 타이틀이 나타나는 것을 볼 수 있다.

내비게이션이 화면으로 지정되었을 때 헤더의 타이틀을 변경하는 방법에 대해 알아보자

MainTab 내비게이션은 MainStack 내비게이션의 화면으로 사용되었기 때문에 다른 화면들과 마찬가지로 props를 통해 navigation과 route를 전달 받는다. 일반적인 화면과 달리 MainTab 내비게이션처럼 Navigator 컴포넌트가 화면으로 사용되는 경우 route에 state가 추가적으로 포함되어 전달된다. route에 포함된 state는 화면의 내비게이션 상태를 확인할 수 있으며 아래와 같은 값들을 갖고 있다.

```
{
  "index" : 0,
  "routeNames" : [
    "Channel List",
    "Profile",
  ],
  "type" : "tab"
  ...
}
```

index는 현재 렌더링되는 화면의 인덱스이며 Screen 컴포넌트가 사용된 순서대로 0부터 지정된다. MainTab 내비게이션의 경우 첫 번째 하위 컴포넌트인 채널 목록 화면이 0, 프로필 화면이 1이 된다.

routeNames는 화면으로 사용되는 Navigator 컴포넌트에서 Screen 컴포넌트들의 name 속성을 배열로 갖고 있다. MainTab 내비게이션은 Screen 컴포넌트로 채널 목록 화면과 프로필 화면을 갖고 있고, 각 컴포넌트의 name으로 사용되는 “Channel List”와 “Profile”을 배열로 갖고 있다.

type은 현재 화면으로 사용되는 Navigator 컴포넌트의 타입이며, MainTab 내비게이션은 탭 내비게이션이기 때문에 “tab”을 값으로 갖는다. 만약 스택 내비게이션으로 만든 Navigator 컴포넌트인 경우 값으로 “stack”을 갖는다.

route의 state로 전달되는 값들을 이용해 현재 렌더링되는 화면을 확인하고 navigation을 이용해서 헤더의 타이틀이 화면의 이름으로 나타나도록 수정해보자.

```
JS Profile.js JS MainTab.js X JS MainStack.js JS theme.js JS Input.js
src > navigations > JS MainTab.js > [X] MainTab > [X] headerRight
20
21 v const MainTab=({ navigation, route })=>{
22   const theme = useContext(ThemeContext);
23
24   const title = getFocusedRouteNameFromRoute(route) ?? 'Channels';
25   navigation.setOptions({
26     headerTitle: title,
27     headerRight: () =>
28     title === 'Channels' && [
29       <MaterialIcons
30         name="add"
31         size={26}
32         style={{ margin: 10 }}
33         onPress={() => navigation.navigate('Channel Creation')}
34       />
35     ],
36   });
37   return(
38     <Tab.Navigator
39     tabBarOptions={{
40       activeTintColor: theme.tabActiveColor,
41       inactiveTintColor: theme.tabInactiveColor,
42     }}
43   >
44     <Tab.Screen name="Channels" component={Channellist}
45     options={{
46       tabBarIcon: ({ focused }) =>
47       TabBarIcon({
48         focused,
49         name: focused ? 'chat-bubble' : 'chat-bubble-outline',
50       }),
51     }}
52   />
53     <Tab.Screen name="Profile" component={Profile}
54     options={{
55       tabBarIcon: ({ focused }) =>
56       TabBarIcon({
57         focused,
58         name: focused ? 'person' : 'person-outline',
59       }),
60     }}
61   />
62 )
```

채널 목록 화면의 name을 Channels로 변경했다. 버전9으로 변경되어서 useEffect없이 코드를 작성했다.

채널 생성 화면

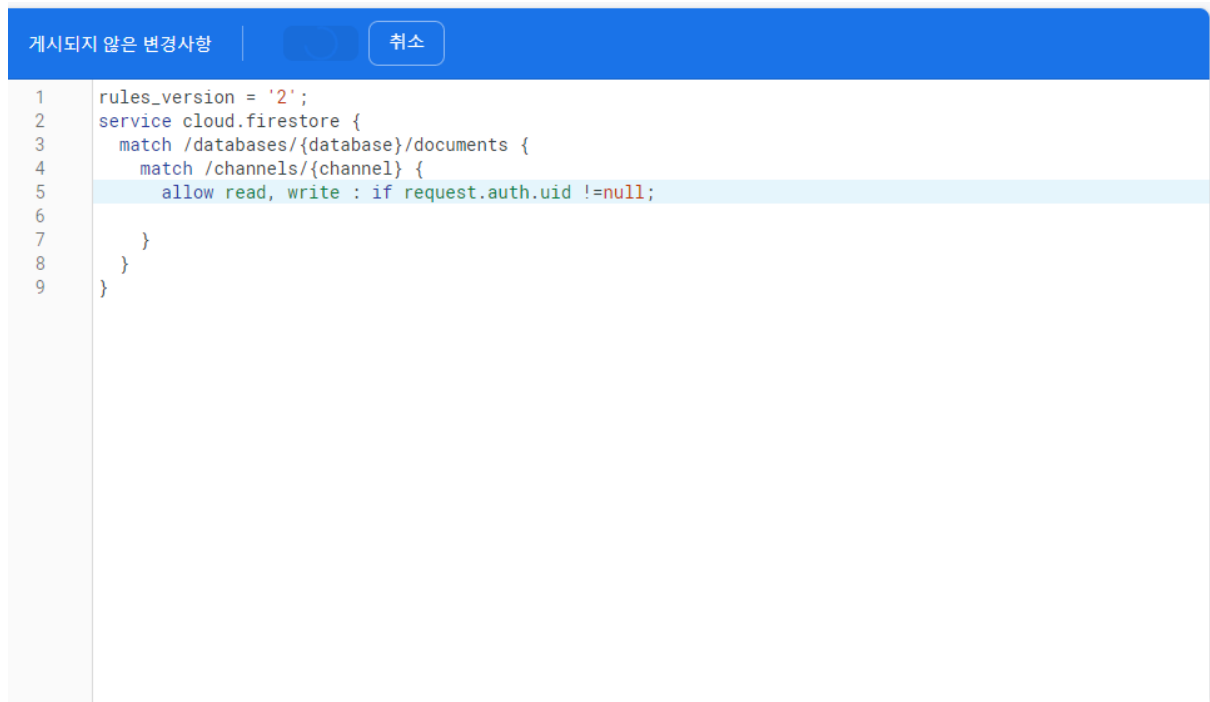
데이터베이스

서버를 구축하지 않기 때문에 채널 데이터를 관리하기 위해 파이어베이스의 데이터베이스를 활용한다. 파이어베이스에서 제공하는 파이어스토어는 NoSQL 문서 중심의 데이터베이스로 SQL 데이터베이스와 달리 테이블이나 행이 없고 컬렉션(collection), 문서(document), 필드(field)로 구성된다. 컬렉션은 문서의 컨테이너 역할을 하며 모든 문서를 항상 컬렉션에 저장되어야 한다. 문서는 파이어스토어의 저장 단위로 값이 있는 필드를 갖는다

문서의 가장 큰 특징은 컬렉션을 필드로 가질 수 있다는 점이다. 파이어스토어는 일반적인 데이터베이스와 달리 데이터베이스의 내용이 수정되면 실시간으로 변경된 내용을 알 수 있다는 특징을 갖고 있다.

그림 p.407

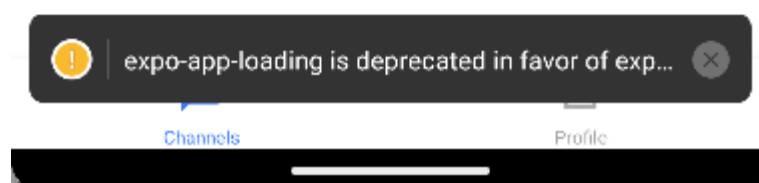
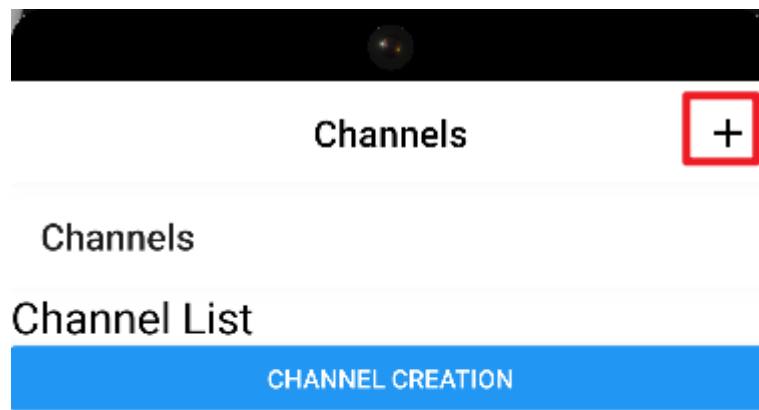
컬렉션과 문서는 항상 유일한 id를 갖고 있어야 한다는 규칙이 있다. 여기서는 channels라는 id를 가진 하나의 컬렉션을 만들고 생성되는 채널들을 channels 컬렉션에 문서로 저장하겠다. 파이어스토어는 채널 생성 시 id를 지정하지 않으면 자동으로 중복되지 않는 id를 생성해서 문서의 id로 이용한다. 따라서 자동으로 생성되는 문서의 id를 이용해 채널문서 id가 중복되지 않도록 관리한다. 마지막으로 인증에 성공한 사용자만 데이터베이스를 읽거나 쓸 수 있도록 데이터베이스의 보안 규칙을 아래와 같이 수정한다.



채널 생성 버튼

채널 목록 화면의 헤더에 채널 생성 화면으로 이동할 수 있는 채널 생성 버튼을 만들자.

```
const title = getFocusedRouteNameFromRoute(route) ?? 'Channels';
navigation.setOptions({
  headerTitle: title,
  headerRight: () =>
    title === 'Channels' && (
      <MaterialIcons
        name="add"
        size={26}
        style={{ margin: 10 }}
        onPress={() => navigation.navigate('Channel Creation')}
      />
    ),
});
```



채널 생성 버튼은 채널 목록 화면의 헤더에만 나타나도록 state의 index값에 따라 렌더링 여부가 결정되도록 만들었다.

채널 생성

채널 생성화면을 수정하고 데이터베이스에 채널 문서를 생성하는 기능을 만들어보자. 채널 생성 화면은 앞에서 만든 컴포넌트들을 이용하면 쉽고 빠르게 만들 수 있다.

```
JS ChannelCreation.js X
src > screens > JS ChannelCreation.js > [🔍] ErrorText
1  import React, {useState, useRef, useEffect} from "react";
2  import styled from "styled-components";
3  import { Input, Button } from "../components";
4  import { KeyboardAwareScrollView } from "react-native-keyboard-aware-scroll-view";
5
6  const Container = styled.View`
7    flex : 1;
8    background-color : ${({theme}) => theme.background};
9    justify-content : center;
10   align-items : center;
11   padding : 0 20px;
12 `;
13 const ErrorText = styled.Text`
14   align-items : flex-start;
15   width : 100%;
16   height : 20px;
17   margin-bottom : 10px;
18   line-height : 20px;
19   color : ${({theme})=> theme.errorText};
20 `;
21 const ChannelCreation = () =>{
22   const [title, setTitle] = useState('');
23   const [description, setDescription] = useState('');
24   const descriptionRef = useRef();
25   const [errorMessage, setErrorMessage] = useState('');
26   const [disabled, setDisabled] = useState(true);
27   useEffect(()=> {
28     setDisabled(!(title && !errorMessage));
29   }, [title, description, errorMessage]);
30 }
```



```

31   const _handleTitleChange = title => {
32     setTitle(title);
33     setErrorMessage(title.trim()? '' : 'Please enter the title');
34   };
35   const _handleCreateButtonPress = () => {};
36   return (
37     <KeyboardAwareScrollView
38       contentContainerStyle={{ flex : 1}}
39       extraScrollHeight= {20}
40     >
41       <Container>
42         <Input
43           label="Title"
44           value={title}
45           onChangeText={_handleTitleChange}
46           onSubmitEditing={() => {
47             setTitle(title.trim());
48             descriptionRef.current.focus();
49           }}
50           onBlur={() => setTitle(title.trim())}
51           placeholder="Title"
52           returnKeyType="next"
53           maxLength={20}
54         />

```

```

55     <Input
56     ref={descriptionRef}
57     label="Description"
58     value={description}
59     onChangeText={text => setDescription(text)}
60     onSubmitEditing={() => {
61       setDescription(description.trim());
62       _handleCreateButtonPress();
63     }}
64     onBlur={() => setDescription(description.trim())}
65     placeholder = "Description"
66     returnKeyType="done"
67     maxLength={40}
68   />
69   <ErrorText>{errorMessage}</ErrorText>
70   <Button
71   title="Channel"
72   onPress={_handleCreateButtonPress}
73   disabled={disabled}
74   />
75 </Container>
76 </KeyboardAwareScrollView>
77 );
78 };
79
80 export default ChannelCreation;

```

← Channel Creation

Title

Description

Channel

firebase.js 파일에서 데이터베이스에 채널을 생성하는 함수를 만들자.

```
JS ChannelCreation.js JS firebasejs X
src > utils > JS firebasejs > [⌘] uploadImage
1  import { initializeApp } from 'firebase/app';
2  import {
3    getAuth,
4    signInWithEmailAndPassword,
5    createUserWithEmailAndPassword,
6    signOut,
7    updateProfile,
8  } from 'firebase/auth';
9  import { getFirestore, collection, doc, setDoc } from 'firebase/firestore';
10 import { getDownloadURL, getStorage, ref, uploadBytes } from 'firebase/storage';
11 import config from '../../firebase.json';
12
13 export const app = initializeApp(config);
14
15 const auth = getAuth(app);
16
17 export const signin = async ({ email, password }) => {
18   const { user } = await signInWithEmailAndPassword(auth, email, password);
19   return user;
20 };
21
22
23 export const signup = async ({ name, email, password, photoUrl }) => {
24   const { user } = await createUserWithEmailAndPassword(auth, email, password);
25   const photoURL = await uploadImage(photoUrl);
26   await updateProfile(auth.currentUser, { displayName: name, photoURL });
27   return user;
28 };
29
```

```
JS ChannelCreation.js    JS firebase.js X
src > utils > JS firebase.js > [x] uploadImage
33  const uploadImage = async uri => {
34    if (uri.startsWith('https')) {
35      return uri;
36    }
37
38    const response = await fetch(uri);
39    const blob = await response.blob();
40
41    const { uid } = auth.currentUser;
42    const storage = getStorage(app);
43    const storageRef = ref(storage, `/profile/${uid}/photo.png`);
44    await uploadBytes(storageRef, blob, {
45      contentType: 'image/png',
46    });
47
48    return await getDownloadURL(storageRef);
49  };
50
51  export const signout = async () => {
52    await signOut(auth);
53    return {};
54  };
55
56  export const getCurrentUser = () => {
57    const { uid, displayName, email, photoURL } = auth.currentUser;
58    return { uid, name: displayName, email, photoUrl: photoURL };
59  };
60
61  export const updateUserInfo = async photo => {
62    const photoUrl = await uploadImage(photo);
63    await updateProfile(auth.currentUser, { photoUrl });
64    return photoUrl;
65  };
66
```

```

68  const db = getFirestore(app);
69
70  export const createChannel = async ({ title, description }) => {
71    const channelCollection = collection(db, 'channels');
72    const newChannelRef = doc(channelCollection);
73    const id = newChannelRef.id;
74    const newChannel = {
75      id,
76      title,
77      description,
78      createdAt: Date.now(),
79    };
80    await setDoc(newChannelRef, newChannel);
81    return id;
82  };

```

컬렉션 문서를 생성할 때 id를 지정하지 않으면 파이어스토어에서 자동으로 중복되지 않는 id를 생성해 문서의 id로 사용한다. 자동으로 생성된 문서의 id는 문서의 필드로도 저장하고, 사용자에게 입력받은 채널의 제목과 설명을 필드로 사용한다. 마지막으로 채널이 생성된 시간을 함수가 호출된 시점의 타임스탬프로 저장하도록 저장하자.

작성된 함수를 이용해서 채널 생성 화면을 아래 코드처럼 수정하자.

```

JS ChannelCreation.js X JS Channel.js JS firebase.js
src > screens > JS ChannelCreation.js > [0] Container
1  import React, {useState, useRef, useEffect, useContext} from "react";
2  import { Alert } from "react-native";
3  import {ProgressContext} from '../contexts';
4  import { createChannel } from "../utils/firebase";
5  import styled from "styled-components";
6  import { KeyboardAwareScrollView } from "react-native-keyboard-aware-scroll-view";
7
8
9
10 const Container = styled.View`
11     flex : 1;
12     background-color : ${({theme}) => theme.background};
13     justify-content : center;
14     align-items : center;
15     padding : 0 20px;
16 `;
17 const ErrorText = styled.Text`
18     align-items : flex-start;
19     width : 100%;
20     height : 20px;
21     margin-bottom : 10px;
22     line-height : 20px;
23     color : ${({theme})=> theme.errorText};
24 `;
25 const ChannelCreation = ({ navigation }) =>{
26     const {spinner} = useContext(ProgressContext);
27     const [title, setTitle] = useState('');
28     const [description, setDescription] = useState('');
29     const descriptionRef = useRef();
30     const [errorMessage, setErrorMessage] = useState('');
31     const [disabled, setDisabled] = useState(true);
32

```

```
JS ChannelCreation.js X JS Channel.js JS firebase.js
src > screens > JS ChannelCreation.js > [🔍] Container
33     useEffect(()=> {
34       |   setDisabled(!(title && !errorMessage));
35     }, [title, description, errorMessage]);
36
37     const _handleTitleChange = title => {
38       |   setTitle(title);
39       |   setErrorMessage(title.trim()? '' : 'Please enter the title');
40     };
41     const _handleCreateButtonPress =async () => {
42       |   try{
43       |     spinner.start();
44       |     const id = await createChannel({title, description});
45       |     navigation.replace('channel', {id, title});
46       |   }catch (e){
47       |     |   Alert.alert('Creation Error', e.message);
48       |   } finally {
49       |     |   spinner.stop();
50       |   }
51   };
52   return (
53     <KeyboardAwareScrollView
54       |   contentContainerStyle={{ flex : 1}}
55       |   extraScrollHeight= {20}
56     >
```



```

JS ChannelCreation.js X JS Channel.js JS firebase.js
src > screens > JS ChannelCreation.js > [0] Container
56 >
57 <Container>
58   <Input
59     label="Title"
60     value={title}
61     onChangeText={_handleTitleChange}
62     onSubmitEditing={() => {
63       setTitle(title.trim());
64       descriptionRef.current.focus();
65     }}
66     onBlur={() => setTitle(title.trim())}
67     placeholder="Title"
68     returnKeyType="next"
69     maxLength={20}
70   />
71   <Input
72     ref={descriptionRef}
73     label="Description"
74     value={description}
75     onChangeText={text => setDescription(text)}
76     onSubmitEditing={() => {
77       setDescription(description.trim());
78       _handleCreateButtonPress();
79     }}
80     onBlur={() => setDescription(description.trim())}
81     placeholder = "Description"
82     returnKeyType="done"
83     maxLength={40}
84   />

```

```

85   <ErrorText>{errorMessage}</ErrorText>
86   <Button
87     title="Channel"
88     onPress={_handleCreateButtonPress}
89     disabled={disabled}
90   />
91 </Container>
92 </KeyboardAwareScrollView>
93 );
94 };
95
96 export default ChannelCreation;

```

채널 생성이 완료되면 채널 생성 화면을 남겨놓은 상태에서 생성된 채널로 이동하는 것이 아니라, 채널 생성 화면을 제거하고 새로 생성된 채널로 이동하는 것이 일반적이다. 채널 생성 화면에서도 동일하게 동작하도록 navigation의 replace 함수를 이용했다. replace 함수는 navigate 함수처럼 화면을 이동하지만, 현재 화면을 스택에 유지하지 않고 새로운 화면과 교체하면서 화면을 이동한다는 특징이 있다.

채널이 생성되는 동안 사용자의 추가 행동을 방지하기 위해 ProgressContext를 이용하여 Spinner 컴포넌트가 렌더링되도록 하고, 채널 생성이 완료되면 채널 화면으로 이동하면서 현재 입장하는 채널의 id와 제목을 params로 함께 전달했다.

이제 채널 화면에서 params로 전달되는 내용을 확인할 수 있도록 아래처럼 코드를 수정하자.

```
JS ChannelCreation.js JS Channel.js X JS firebase.js
src > screens > JS Channel.js > ...
1  import React from "react";
2  import styled from "styled-components";
3  import {Text} from 'react-native';
4
5  const Container = styled.View`
6    flex : 1;
7    background-color : ${({theme}) => theme.background};
8  `;
9
10 const Channel =( route )=>{
11   return(
12     <Container>
13       <Text style={{ fontSize : 24 }}>ID : {route.params?.id}</Text>
14       <Text style={{ fontSize : 24 }}>Title : {route.params?.title}</Text>
15     </Container>
16   );
17 };
18
19 export default Channel;
```

Channel Creation

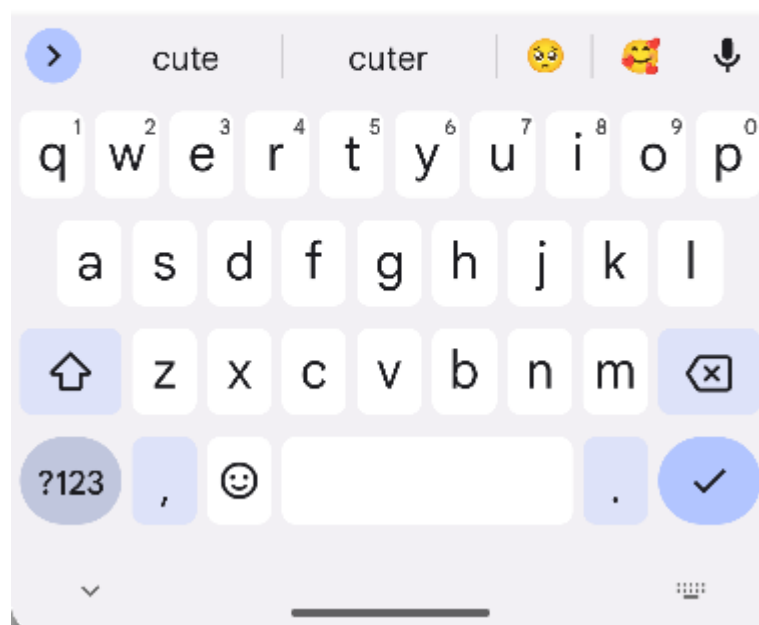
Title

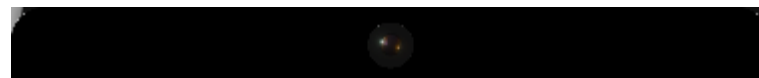
Hongsi

Description

hongsi is cute

Create





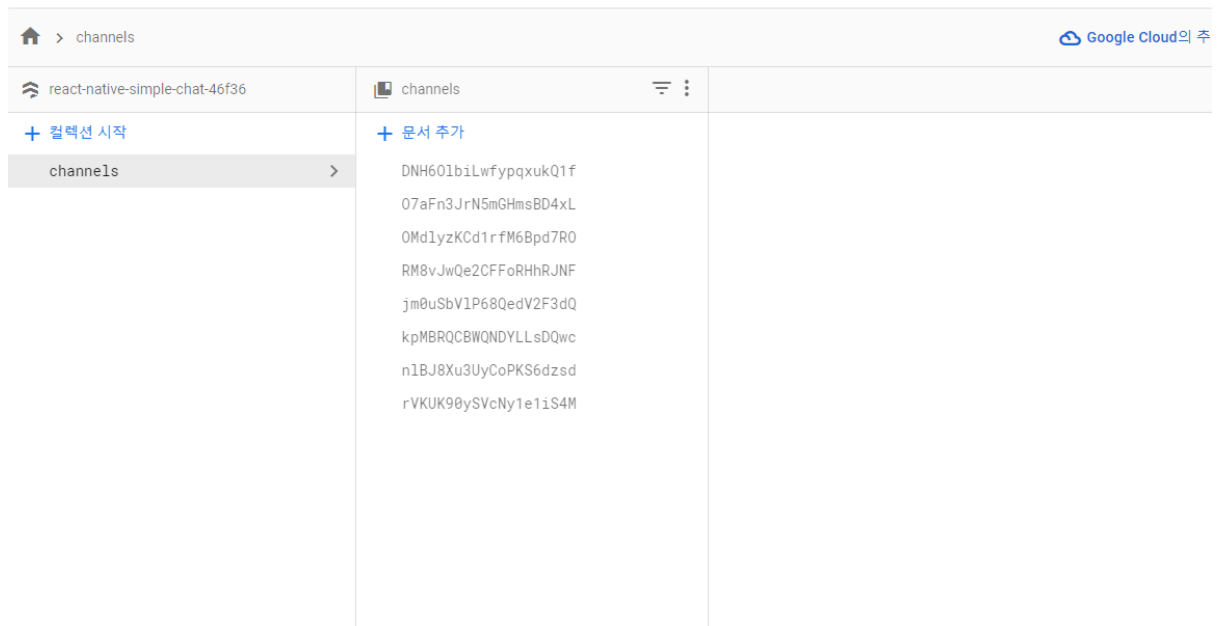
Channel

ID : jm0uSbVIP68QedV2F3dQ

Title : Hongsi



채널을 생성하면 생성된 채널의 정보와 함께 채널 화면으로 이동하고, 뒤로 가기 버튼을 클릭하면 채널 생성 화면이 아니라 채널 목록 화면으로 이동하는 것을 확인할 수 있다. 파이어베이스 콘솔에서도 생성된 채널 문서와 내용을 확인할 수 있다.



채널 목록 화면

생성된 채널을 보여주는 채널 목록 화면을 만들자.

많은 양의 채널을 목록으로 사용자에게 보여줄 수 있으며, 채널이 새로 생성되면 자동으로 채널 목록에 추가되도록 화면을 만들어 보자.

FlatList 컴포넌트

지금까지 많은 양의 데이터를 렌더링할 때 ScrollView 컴포넌트를 이용해 화면이 넘어가도 스크롤이 생성되어 확인할 수 있도록 만들었다.

ScrollView 컴포넌트와 같은 역할을 하는 컴포넌트로 리액트 네이티브에서 제공하는 FlatList 컴포넌트가 있다. 두 컴포넌트 모두 많은 양의 데이터를 목록으로 보여주는 상황에서 자주 사용되고, 렌더링된 내용이 화면을 넘어가면 스크롤이 생성된다는 공통점이 있다.

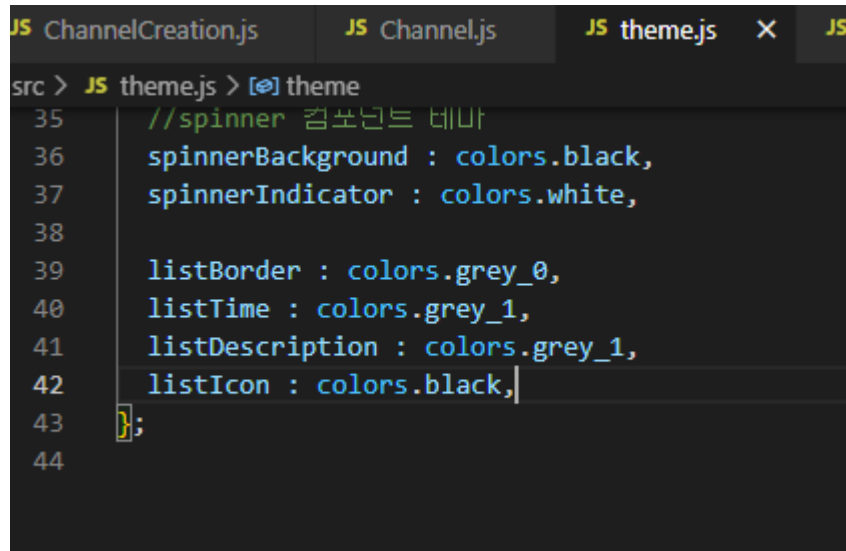
ScrollView 컴포넌트는 렌더링해야 하는 모든 데이터를 한번에 렌더링하므로 렌더링해야 하는 데이터의 양을 알고 있을 때 사용하는 것이 좋다. 렌더링하는 데이터가 매우 많을 경우 한번에 모든 데이터를 렌더링하면 렌더링 속도가 느려지고 메모리 사용량이 증가하는 등 성능이 저하된다는 문제가 있다.

반면에 FlatList 컴포넌트는 화면에 적절한 양의 데이터만 렌더링하고 스크롤의 이동에 맞춰 필요한 부분을 추가적으로 렌더링하는 특징이 있다. 이런 특징 덕분에 데이터의 길이가 가변

적이고 양을 예측할 수 없는 상황에서 사용하기 좋다.

채널 목록 화면에서 FlatList 컴포넌트를 이용해 생성된 채널들을 렌더링하도록 만들자.

채널 목록 화면에서 사용할 색을 theme.js파일에 정의한다.



```
src > JS theme.js > [⌘] theme
35 //spinner 컴포넌트 테마
36 spinnerBackground : colors.black,
37 spinnerIndicator : colors.white,
38
39 listBorder : colors.grey_0,
40 listTime : colors.grey_1,
41 listDescription : colors.grey_1,
42 listIcon : colors.black,
43 };
44
```

FlatList 컴포넌트를 사용하려면 3개의 속성을 지정해야한다. 먼저 렌더링할 항목의 데이터를 배열로 전달해야 하고, 전달된 배열의 항목을 이용해 항목을 렌더링하는 함수를 작성해야 한다. 마지막으로 각 항목에 키를 추가하기 위해 고유한 값을 반환하는 함수를 전달해야 한다. 아직 충분히 많은 채널이 준비된 것은 아니므로 임의로 1000개의 채널 데이터를 생성해 채널 목록 화면을 만들어보자.

JS ChannelList.js X

src > screens > JS ChannelList.js > ...

```
1  import React, {useContext} from "react";
2  import { FlatList } from "react-native-gesture-handler";
3  import styled, {ThemeContext} from "styled-components";
4  import {MaterialIcons} from '@expo/vector-icons';
5  import {Text, Button} from 'react-native';
6
7  const Container = styled.View`
8    flex : 1;
9    background-color : ${({theme}) => theme.background};
10 `;
11
12 const ItemContainer = styled.TouchableOpacity`
13   flex-direction : row;
14   align-items : center;
15   border-bottom-width : 1px;
16   border-color : ${({theme}) => theme.listBorder};
17   padding : 15px 20px;
18 `;
19
20 const ItemTextContainer = styled.View`
21   flex : 1;
22   flex-direction : column;
23 `;
24
25 const ItemTitle = styled.Text`
26   font-size : 20px;
27   font-weight : 600;
28 `;
29
30 const ItemDescription = styled.Text`
31   font-size : 16px;
32   margin-top : 5px;
33   color : ${({theme}) => theme.listDescription};
34 `;
35
36 const ItemTime = styled.Text`
37   font-size : 12px;
38   color : ${({theme}) => theme.listTime};
39 `;
```

```
35   `;
36   const channels = [];
37   for (let idx = 0; idx < 1000; idx++){
38     channels.push ({
39       id : idx,
40       title : `title ${idx}`,
41       description : `description ${idx}`,
42       createdAt : idx,
43     });
44   }
45
46   const Item = ({ item : {id, title, description, createdAt}, onPress}) => {
47     const theme = useContext(ThemeContext);
48     console.log(`Item : ${id}`);
49
50     return(
51       <ItemContainer onPress={() => onPress({id, title})}>
52         <ItemTextContainer>
53           <ItemTitle>{title}</ItemTitle>
54           <ItemDescription>{description}</ItemDescription>
55         </ItemTextContainer>
56         <ItemTime>{createdAt}</ItemTime>
57         <MaterialIcons
58           name="keyboard-arrow-right"
59           size={24}
60           color={theme.listIcon}
61         />
62       </ItemContainer>
63     );
64   };
```

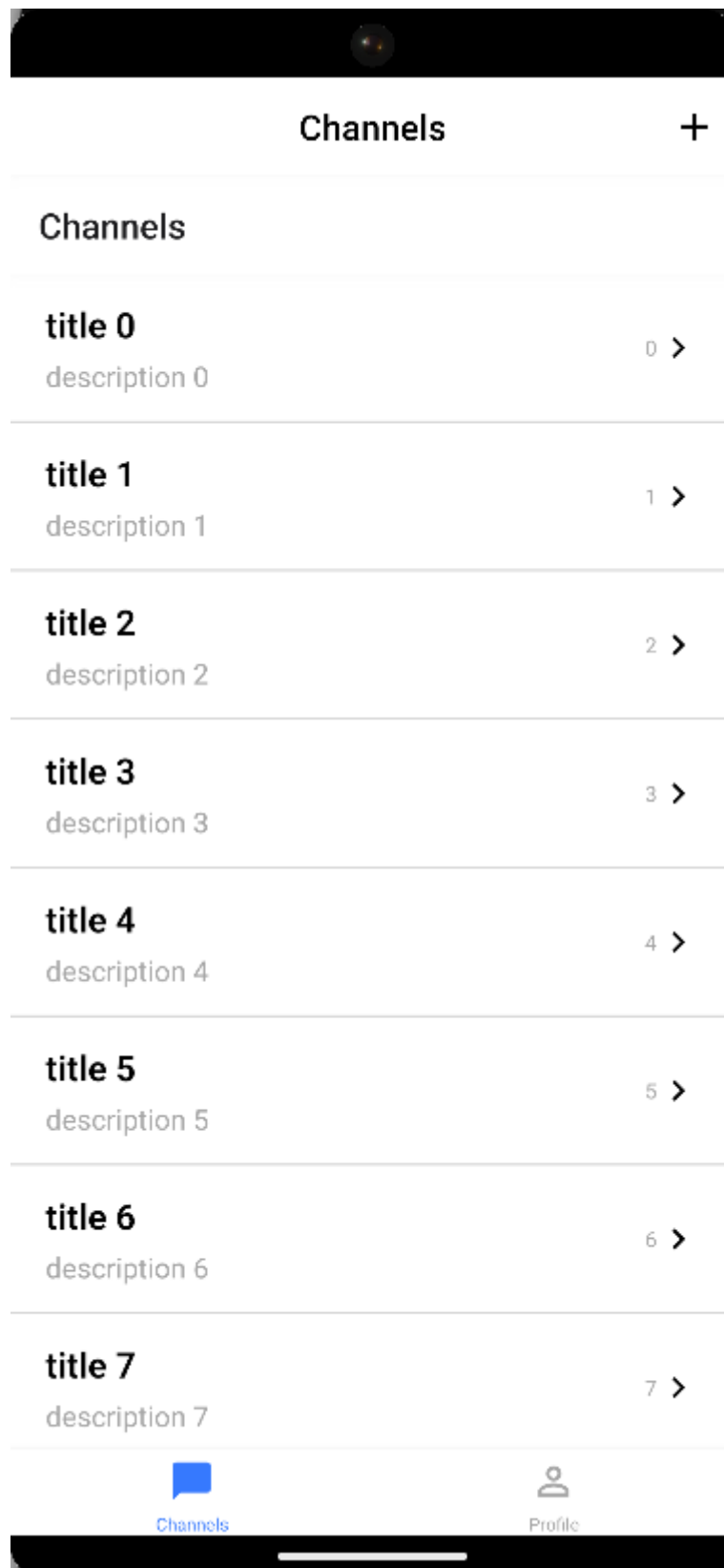


```

66  const ChannelList = ({navigation}) =>{
67    const _handleItemPress = params => {
68      navigation.navigate('Channel', params);
69    };
70    return(
71      <Container>
72        <FlatList
73          keyExtractor={item => item['id'].toString()}
74          data={channels}
75          renderItem={({item}) => (
76            <Item item={item} onPress={_handleItemPress} />
77          )}
78        />
79      </Container>
80    );
81  };
82  export default ChannelList;

```

생성된 임의의 데이터를 FlatList 컴포넌트에 항목으로 사용할 데이터로 설정했다. renderItem에 작성되는 함수는 파라미터로 항목의 데이터를 가진 item이 포함된 객체가 전달된다. 파라미터로 전달되는 데이터를 이용해서 각 항목의 내용을 렌더링하고 클릭 시 채널 화면으로 이동하도록 만들었다. 마지막으로 각 항목의 id값을 키로 이용하도록 keyExtractor를 설정했다.



windowSize

item 함수에 작성된 로그를 터미널에서 확인하면 FlatList 컴포넌트의 특징 때문에 우리가 만든 1000개의 데이터 중 id 0번부터 100번까지 101개만 렌더링 된것을 알 수 있다. 화면에는 9개의 항목이 보이므로 화면을 넘어가는 데이터 92개가 추가로 렌더링 된다는 것을 알 수 있다.

※ 화면의 크기에 따라 렌더링 되는 항목의 수와 터미널을 통해 확인되는 렌더링된 데이터의 양에 차이가 있을 수 있다.

그러면 왜 101개만 렌더링 되었을까?

FlatList 컴포넌트에서 렌더링되는 데이터의 수는 `windowSize` 속성에 의해 결정된다. `windowSize`의 기본값은 21이고, 이 값은 현재 화면(1)과 현재 화면보다 앞쪽에 있는 데이터(10), 그리고 현재 화면 보다 뒤쪽에 있는 데이터(10)를 의미한다. 예를들어 한 화면에 10개의 항목이 렌더링되고 현재 화면의 앞뒤로 충분히 많은 데이터가 있다고 가정하자. 현재 화면보다 이전 데이터 중에서 화면 10개만큼 렌더링할 수 있는 100개의 데이터를 렌더링하고, 이후 데이터 중에서도 화면 10개만큼 렌더링 할 수 있는 100개의 데이터를 렌더링하여 총 210개의 데이터를 렌더링한다

현재 화면 (10 items) + 이전 데이터 (10 items x 10 screens) + 이후 데이터 (10 items x 10 screens) = 210

화면은 가장 앞쪽 데이터 이므로 이전 화면을 위해 렌더링할 데이터는 없으며 이후 화면들을 위한 데이터 (9items x 10screens)를 렌더링한다. 추가적으로 가장 아래에 약 20%정도 보이는 10번째 항목을 0.2개로 보면 총 101.2 개의 데이터가 렌더링되어야 하므로 101개의 데이터가 렌더링되는 것이다.

현재 화면(9.2 items) + 이전 데이터(0 items) + 이후 데이터(9.2 items x 10 screens) = 101.2

화면을 스크롤하면서 터미널을 통해 렌더링되는 데이터를 확인해보자

만약 렌더링되는 데이터의 양을 조절하고 싶다면 `windowSize`의 값을 원하는 값으로 설정한다.

windowSize의 값을 작은 값으로 변경하면 렌더링 되는 데이터가 줄어들어 메모리의 소모비를 줄이고 성능을 향상시킬 수 있지만, 빠르게 스크롤하는 상황에서 미리 렌더링되지 않은 부분은 순간적으로 빈 내용이 나타날 수 있다는 단점이 있다. 여기서는 채널 목록 화면에서 windowSize의 값을 3으로 변경하여 현재 화면과 앞뒤로 한 화면만큼만 렌더링 되도록 조절하자

```
JS ChannelList.js X
src > screens > JS ChannelList.js > ChannelList
64   };
65
66   const ChannelList = ({navigation}) => {
67     const _handleItemPress = params => {
68       navigation.navigate('Channel', params);
69     };
70     return(
71       <Container>
72         <FlatList
73           keyExtractor={item => item['id'].toString()}
74           data={channels}
75           renderItem={({item}) => (
76             <Item item={item} onPress={_handleItemPress} />
77           )}
78           windowSize={3}
79         />
80       </Container>
81     );
82   };
83   export default ChannelList;
```

React.memo

windowSize를 수정해서 렌더링되는 양을 줄였지만 스크롤을 이동해보면 비효율적인 부분이 보인다.

스크롤이 이동하면 windowSize 값에 맞춰 현재 화면과 이전 및 이후 데이터를 렌더링하는 것이 맞지만, 이미 렌더링된 항목도 다시 렌더링되는 것을 확인할 수 있다. React.memo를 이용하면 이런 비효율적인 반복 작업을 해결할 수 있다. React.memo는 useMemo Hook 함수와 매우 흡사하지만, 불필요한 함수의 재연산을 방지하는 useMemo와 달리 React.memo는 컴포넌트의 리렌더링을 방지한다는 차이가 있다.

```
JS ChannelList.js X
src > screens > JS ChannelList.js > Item > React.memo() callback
45
46 const Item = React.memo(
47   ({ item : {id, title, description, createdAt}, onPress}) => {
48     const theme = useContext(ThemeContext);
49     console.log(`Item : ${id}`);
50
51     return(
52       <ItemContainer onPress={() => onPress({id, title})}>
53         <ItemTextContainer>
54           <ItemTitle>{title}</ItemTitle>
55           <ItemDescription>{description}</ItemDescription>
56         </ItemTextContainer>
57         <ItemTime>{createdAt}</ItemTime>
58         <MaterialIcons
59           name="keyboard-arrow-right"
60           size={24}
61           color={theme.listIcon}
62         />
63       </ItemContainer>
64     );
65   }
66 );
```

React.memo를 이용해 컴포넌트를 감싸는 것으로 간단히 적용할 수 있다. 이제 Item컴포넌트는 props가 변경될 때까지 리렌더링되지 않는다. 결과를 확인해보면 스크롤이동을 해도 이미 렌더링된 컴포넌트의 리렌더링이 발생하지 않는 것을 볼 수 있다.

채널 데이터 수신

채널 목록 화면에서 파이어베이스의 데이터베이스로부터 데이터를 받아 채널 목록을 렌더링하도록 하자

채널 목록 화면에서 렌더링할 데이터를 생성하자. 필수는 아니지만, 테스트를 위해 일정 수준 이상의 데이터가 준비된 상태에서 진행하는 것이 좋다.

앞에서 만든 채널 생성 화면을 이용하거나 파이어베이스 콘솔의 데이터 베이스 메뉴에서 문서 추가를 이용하면 데이터를 추가할 수 있다. 파이어베이스 콘솔에서 데이터를 추가하는 경우 title과 description 필드의 타입을 문자열로 입력하고, 문서의 id는 자동으로 생성되는 값

을 사용하여 그 값을 문서의 id필드에 저장해주자. 마지막으로 createdAt 필드의 타입은 숫자 타입으로 지정해야 하며 정상적인 값이 추가되도록 자바스크립트 함수를 이용한다.

채력 목록으로 사용할 데이터 준비가 완료되면 채널 목록 화면에서 데이터베이스의 채널 데이터를 받아오도록 만들자.

```
import { app } from '../utils/firebase';  
import {  
  getFirestore,  
  collection,  
  onSnapshot,  
  query,  
  orderBy,  
} from 'firebase/firestore';
```

src > screens > JS ChannelList.js > [🔍] ItemTitle

```
1  import React, {useContext, useState, useEffect} from "react";
2  import { FlatList } from "react-native-gesture-handler";
3  import styled, {ThemeContext} from "styled-components";
4  import {MaterialIcons} from '@expo/vector-icons';
5  import {Text, Button} from 'react-native';
6  import {DB} from '../utils/firebase'
7
8  const Container = styled.View`
9    flex : 1;
10   background-color : ${({theme}) => theme.background};
11 `;
12
13 const ItemContainer = styled.TouchableOpacity`
14   flex-direction : row;
15   align-items : center;
16   border-bottom-width : 1px;
17   border-color : ${({theme}) => theme.listBorder};
18   padding : 15px 20px;
19 `;
20 const ItemTextContainer = styled.View`
21   flex : 1;
22   flex-direction : column;
23 `;
24 const ItemTitle = styled.Text`
25   font-size : 20px;
26   font-weight : 600;
27 `;
28 const ItemDescription = styled.Text`
29   font-size : 16px;
30   margin-top : 5px;
31   color : ${({theme}) => theme.listDescription};
32 `;
33 const ItemTime = styled.Text`
34   font-size : 12px;
35   color : ${({theme}) => theme.listTime};
36 `;
```

```
JS ChannelList.js X JS ChannelCreation.js
src > screens > JS ChannelList.js > [0] ItemTitle
38   for (let idx = 0; idx < 1000; idx++){
39     channels.push ({
40       id : idx,
41       title : `title ${idx}`,
42       description : `description ${idx}`,
43       createdAt : idx,
44     });
45   }
46
47   const Item = React.memo(
48     ({ item : {id, title, description, createdAt}, onPress}) => {
49     const theme = useContext(ThemeContext);
50     console.log(`Item : ${id}`);
51
52     return(
53       <ItemContainer onPress={() => onPress({id, title})}>
54       <ItemTextContainer>
55         <ItemTitle>{title}</ItemTitle>
56         <ItemDescription>{description}</ItemDescription>
57       </ItemTextContainer>
58       <ItemTime>{createdAt}</ItemTime>
59       <MaterialIcons
60         name="keyboard-arrow-right"
61         size={24}
62         color={theme.listIcon}
63       />
64       </ItemContainer>
65     );
66   }
67 );
68 const Channellist = ({navigation}) =>{
69   // 채널 배열 담기
70   const [channels, setChannels] = useState([]);
71   //DB 연결
72   const db=getFirestore(app);
73
74   useEffect(() => {
75     const collectionQuery = query(
76       collection(db, 'channels'),
77       orderBy('createdAt', 'desc')
78     );
79   });
80 }
```



```
JS ChannelList.js X JS ChannelCreation.js
src > screens > JS ChannelList.js > ChannelList

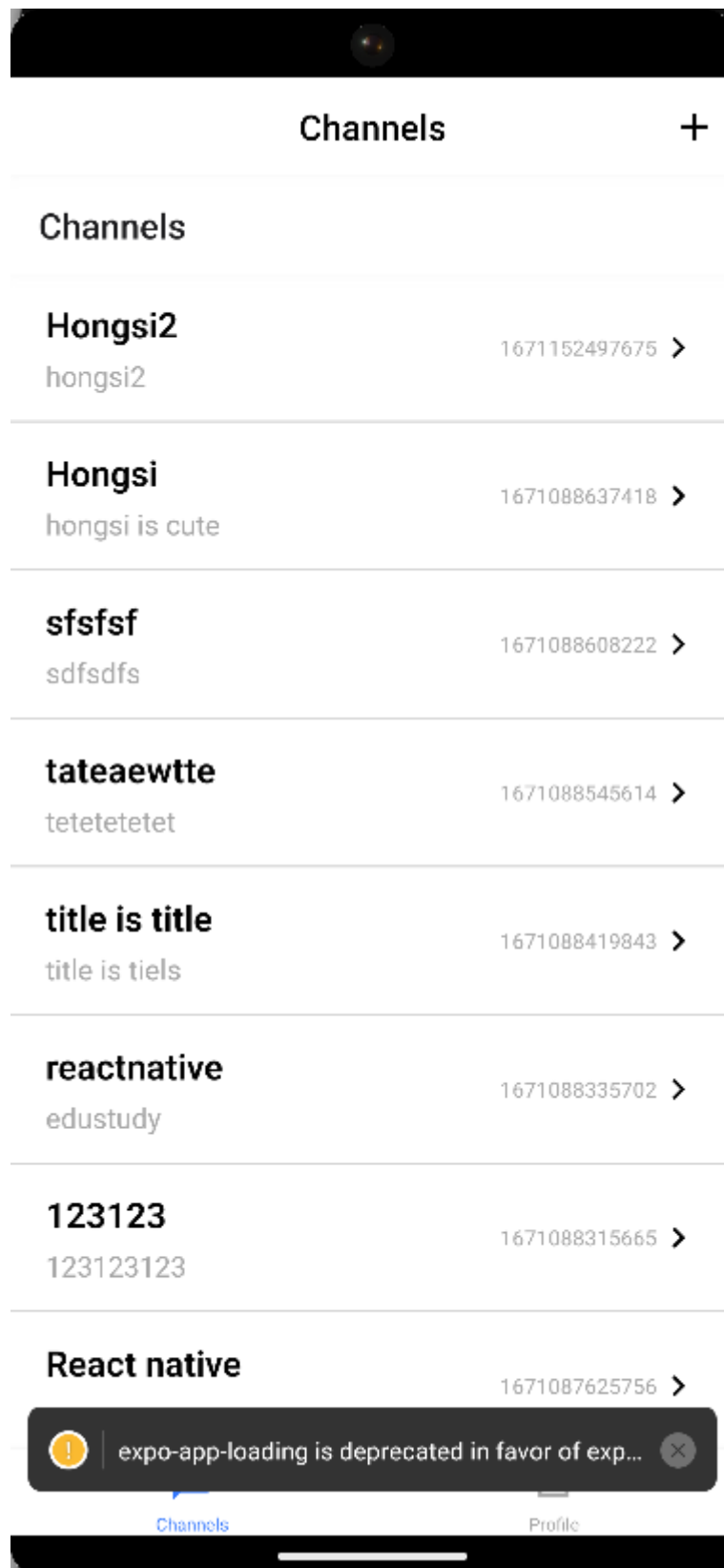
86
87   const unsubscribe = onSnapshot(collectionQuery, snapshot => {
88     const list = [];
89     snapshot.forEach(doc => {
90       list.push(doc.data());
91     });
92     setChannels(list);
93   });
94   return () => unsubscribe();
95
96   }, []);
97   const _handleItemPress = params => {
98     navigation.navigate('Channel', params);
99   };
100
101   return(
102     <Container>
103       <FlatList
104         keyExtractor={item => item['id']}
105         data={channels}
106         renderItem={({item}) => (
107           <Item item={item} onPress={_handleItemPress} />
108         )}
109         windowSize={3}
110       />
111     </Container>
112   );
113 };
114 export default ChannelList;
```

렌더링할 데이터를 데이터베이스에서 받아온 후 `useState` 함수를 이용해서 관리할 `channels`를 생성했고, 테스트를 위해 생성한 임의의 100개 데이터는 삭제했다. 항목의 키도 데이터베이스에서 받아온 채널 문서의 `id`를 이용하면서 타입을 변환하는 코드가 필요 없어져 삭제했다.

`useEffect` 함수를 이용해서 채널 목록 화면이 마운트될 때 `onSnapshot` 함수를 이용하여 데이터베이스에서 데이터를 수신하도록 작성했다. `onSnapshot` 함수는 수신 대기 상태로 있다가 데이터베이스에 문서가 추가되거나 수정될 때마다 지정된 함수가 호출 된다. 최근에 만들어진 채널이 가장 위에 나올 수 있도록 데이터 조회조건으로 `createdAt` 필드값의 내림차순을 설정했다.

채널 목록 화면이 마운트될 때 채널 데이터 수신 대기 상태가 되도록 하고, 화면이 언마운트될 때 수신 대기 중인 상태를 해제하도록 만들었다. 수신 대기 상태를 해제하지 않으면 다시 채널 목록 화면이 마운트될 때 수신 대기 이벤트가 추가되면서 데이터를 중복으로 받는 문제가 발생하므로 주의해야 한다.

버전이 올라감에 따라 명령어가 조금 변했다.



moment 라이브러리

렌더링된 채널 목록을 보면 생성된 시간이 타임스탬프로 나타나 정확한 시간을 알아볼 수 없다. 이번에는 타임스탬프를 사용자가 알아볼 수 있는 시간 형태로 수정해보자

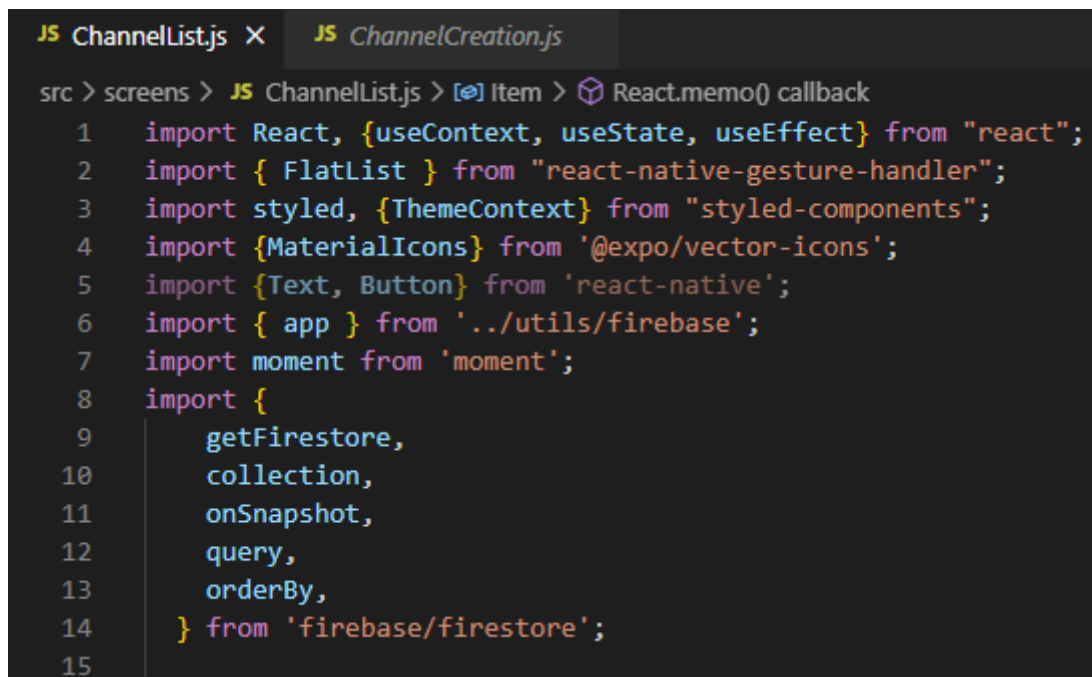
자바스크립트에 내장된 함수를 이용하면 타임스탬프를 우리에게 익숙한 시간이나 날짜 형태로 변경할 수 있다. 하지만 자바스크립트에서 제공하는 기능만으로 기능을 구현하다 보면 조건에 따라 굉장히 복잡해지고 생각하지 못한 버그들도 많이 생긴다

moment 라이브러리를 사용하면 시간 및 날짜와 관련된 함수를 쉽게 작성할 수 있다.

moment 라이브러리를 사용해본 적 없는 사람은 이번기회에 경험해보자. 날짜 관련 라이브러리 중 가장 유명하며 많은 기능을 제공하고 있어 유용하게 사용할 수 있다.

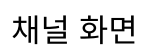
아래 명령어를 통해 moment라이브러리를 설치하자

- `npm install moment (--force)`



```
JS ChannelList.js X JS ChannelCreation.js
src > screens > JS ChannelList.js > [🔍] Item > 📦 React.memo() callback
1  import React, {useContext, useState, useEffect} from "react";
2  import { FlatList } from "react-native-gesture-handler";
3  import styled, {ThemeContext} from "styled-components";
4  import {MaterialIcons} from '@expo/vector-icons';
5  import {Text, Button} from 'react-native';
6  import { app } from '../utils/firebase';
7  import moment from 'moment';
8  import {
9      getFirestore,
10     collection,
11     onSnapshot,
12     query,
13     orderBy,
14   } from 'firebase/firestore';
15
```

```
JS ChannelList.js X JS ChannelCreation.js
src > screens > JS ChannelList.js > [🔍] Item > 📦 React.memo() callback
55 const getDataOrTime = ts => {
56   const now = moment().startOf('day');
57   const target = moment(ts).startOf('day');
58   return moment(ts).format(now.diff(target, 'days') > 0 ? 'MM/DD' : 'HH:mm') ;
59 };
60 const Item = React.memo(
61   ({ item : {id, title, description, createdAt}, onPress}) => {
62     const theme = useContext(ThemeContext);
63     console.log(`Item : ${id}`);
64
65     return(
66       <ItemContainer onPress={() => onPress({id, title})}>
67         <ItemTextContainer>
68           <ItemTitle>{title}</ItemTitle>
69           <ItemDescription>{description}</ItemDescription>
70         </ItemTextContainer>
71         <ItemTime>{getDataOrTime(createdAt)}</ItemTime>
72         <MaterialIcons
73           name="keyboard-arrow-right"
74           size={24}
75           color={theme.listIcon}
76         />
77       </ItemContainer>
78     );
79   }
80 );
```



이번에는 사용자가 메시지를 주고받을 수 있는 채널 화면을 만들어보자

데이터베이스

여기서는 channels 컬렉션 아래에서 채널들을 문서로 관리하고 있다. 각 채널 아래에서 메시지들을 관리하려면 어떻게 해야 할까. 파이어스토어의 문서가 보유한 특징 중 하나인 컬렉션을 가질 수 있다는 점을 활용하여 각 채널 문서에 messages 컬렉션을 만들면 메시지 데이터를 관리할 수 있다. 이렇게 채널별로 발생한 메시지를 모아서 관리하면 채널에서 주고받는 메시지를 편하게 저장하고 불러올 수 있다.

그림 p.428

추가될 데이터베이스의 구조에 맞춰 보안 규칙도 수정해보자. channels 컬렉션의 문서 아래에 있는 컬렉션에 대한 규칙은 데이터베이스의 구조처럼 channels 컬렉션 규칙 안에 해당 규칙을 작성하면 된다.

The screenshot shows the Firebase Security Rules editor. On the left, the 'create' rule is selected for the path '/databases/(default)/documents/channels/ch1/messages/msg1'. The main editor displays the following rule code:

```
1 rules_version = '2';
2 service cloud.firestore {
3   match /databases/{database}/documents {
4
5     match /channels/{channel} {
6       allow read, write : if request.auth.uid != null;
7
8       match /messages/{message} {
9         allow read, write : if request.auth.uid != null;
10      }
11    }
12  }
13 }
```

On the right, a table shows the rule's metadata:

request
... !=null

메신저 데이터

채널 화면도 채널 목록 화면처럼 메시지 데이터의 변화를 실시간으로 전달받기 위해 onSnapshot 함수를 이용하여 수신 대기 상태가 되도록 수정했다. 그리고 채널 화면으로 이

동할 때 route의 params를 통해 전달된 채널의 문서 ID를 이용하여 채널 문서에 있는 messages 컬렉션의 문서 변화에 대해 수신 대기 상태가 되도록 작성했다.

코드

```
JS ChannelList.js JS Channel.js X
src > screens > JS Channel.js > [0] Channel

1  import React, {useState, useEffect, useLayoutEffect} from "react";
2  import styled from "styled-components";
3  import {Text, FlatList} from 'react-native';
4  import {createMessage, getCurrentUser, app} from '../utils/firebase';
5  import {
6    getFirestore,
7    collection,
8    onSnapshot,
9    query,
10   doc,
11   orderBy,
12 } from 'firebase/firestore';
13
14  const Container = styled.View`
15    flex : 1;
16    background-color : ${({theme}) => theme.background};
17  `;
```



```
JS ChannelList.js JS Channel.js X
src > screens > JS Channel.js > [e] Channel
18
19 const Channel = ({ navigation, route }) => {
20   const [messages, setMessages] = useState([]);
21
22   const db = getFirestore(app);
23   useEffect(() => {
24     const docRef = doc(db, 'channels', route.params.id);
25     const collectionQuery = query(
26       collection(db, `${docRef.path}/messages`),
27       orderBy('createdAt', 'desc')
28     );
29     const unsubscribe = onSnapshot(collectionQuery, snapshot => {
30       const list = [];
31       snapshot.forEach(doc => {
32         list.push(doc.data());
33       });
34       setMessages(list);
35     });
36     return () => unsubscribe();
37   }, []);
38   useLayoutEffect(() => {
39     navigation.setOptions({ headerTitle : route.params.title || 'Channel' });
40   }, []);
41   return (
42     <Container>
43       <FlatList
44         keyExtractor={item => item['id']}
45         data={messages}
46         renderItem={({ item }) => (
47           <Text style={{ fontSize : 24 }}>{item.text}</Text>
48         )}
49       />
50     </Container>
51   );
52 };
53
54
55 export default Channel;
```

메시지 데이터도 채널 데이터와 마찬가지로 최신 데이터부터 받아오기 위해 createdAt 필드 값의 내림차순으로 정렬했다. 수신 대기 상태는 언마운트 시 꼭 해제해야 한다는 것을 잊지 말자.

채널 화면의 헤더 타이틀을 채널의 이름이 렌더링되도록 하여 사용자가 대화하는 채널을 인지할 수 있도록 수정했다. 마지막으로 FlatList 컴포넌트를 이용해 메시지가 렌더링되도록 작성했다.



메시지 전송

이번에는 메시지를 전송하는 기능을 만들어보자 먼저 firebase.js 파일에 메시지를 생성하는 함수를 작성하자.

코드

```
JS ChannelList.js X JS Channel.js JS firebase.js X
src > utils > JS firebase.js > ...
84 export const createMessage = async ({ channelId, text }) => {
85   const docRef = doc(db, `channels/${channelId}/messages`, text);
86   await setDoc(docRef, { text, createdAt: Date.now() });
87 };
```

메시지가 저장될 messages 컬렉션이 위치한 채널 문서를 찾기 위해 채널 문서의 ID를 전달 받도록 작성했다. add 함수를 이용하여 문서의 내용만 전달하면 파이어베이스에서 자동으로 문서의 id를 생성하여 적용해준다. 이제 채널 화면에 Input 컴포넌트를 추가하고 입력되는 메시지를 데이터베이스에 저장해보자.

코드

```
JS ChannelList.js JS Channel.js X JS firebase.js
src > screens > JS Channel.js > ...
1  import React, { useState, useEffect, useLayoutEffect, useContext } from 'react';
2  import styled from 'styled-components';
3  import { Alert } from 'react-native';
4  import { Text, FlatList } from 'react-native';
5  import { createMessage, getCurrentUser, app } from '../utils/firebase';
6  import {
7    getFirestore,
8    collection,
9    onSnapshot,
10   query,
11   doc,
12   orderBy,
13 } from 'firebase/firestore';
14 import { Input } from '../components';
15
```

```
JS ChannelList.js JS Channel.js X JS firebase.js
src > screens > JS Channel.js > ...

17
18
19 const Container = styled.View`
20   flex: 1;
21   background-color: ${({theme})=>theme.background};
22 `;
23
24 const Channel = ({ navigation, route }) => {
25   const [messages, setMessages] = useState([]);
26   const [text, setText] = useState('');
27
28   const db = getFirestore(app);
29   useEffect(() => {
30     const docRef = doc(db, 'channels', route.params.id);
31     const collectionQuery = query(
32       collection(db, `${docRef.path}/messages`),
33       orderBy('createdAt', 'desc')
34     );
35     const unsubscribe = onSnapshot(collectionQuery, snapshot => {
36       const list = [];
37       snapshot.forEach(doc => {
38         list.push(doc.data());
39       });
40       setMessages(list);
41     });
42     return () => unsubscribe();
43   }, []);
44
45   useEffect(() => {
46     navigation.setOptions({ headerTitle: route.params.title || 'Channel' });
47   }, []);
48
49   const _handleMessageSend = async messageList => {
50     const newMessage = messageList[0];
51     try {
52       await createMessage({ channelId: route.params.id, message: newMessage });
53     } catch (e) {
54       Alert.alert('Send Message Error', e.message);
55     }
56   };
57 }
```

```
JS ChannelList.js JS Channel.js X JS firebase.js
src > screens > JS Channel.js > ...
57
58   return(
59     <Container>
60       <FlatList
61         keyExtractor={item => item['id']}
62         data={messages}
63         renderItem={({item})=>(
64           <Text style={{fontSize: 24}} >{item.text}</Text>
65         )}
66       />
67       <Input
68         label = "message"
69         value = {text}
70         onChangeText = {text=>setText(text)}
71         onSubmitEditing={()=>createMessage({ channelId: route.params.id, text: text })}
72       />
73     </Container>
74   )
75
76 }
77
78
79 export default Channel;
```

Input 컴포넌트로 사용자에게 메시지를 입력받고 firebase.js 파일에 작성한 createMessage 함수를 이용해서 메시지 문서를 생성하도록 만들었다.



화면

나타나는 경고 메시지는 일단 무시하고 메시지를 생성해보자, 파이어베이스 콘솔에도 정상적으로 생성되며 onSnapshot 함수를 통해 생성된 메시지가 잘 전달되었는지 확인해보자.

🏠 > channels > K2TIU3QE4c851L... > messages > 123			Google Cloud의 추가 기능		
K2TIU3QE4c851LvELGIA		messages	123		
+ 컬렉션 시작		+ 문서 추가	+ 컬렉션 시작		
messages >		123 >	+ 필드 추가		
			createdAt: 1671159446634		
			text: "123"		
+ 필드 추가					
createdAt: 1671152497675					
description: "hongsi2"					
id: "K2TIU3QE4c851LvELGIA"					
title: "Hongsi2"					

버전이 오르고 책이랑 다른 부분이 너무 많아졌다.

text만 넣어도 되는게 text : text로 넣어야 하고...

GiftedChat 컴포넌트

메시지를 주고받는 화면은 일반적인 모바일 화면과 스크롤 방향이 반대이다. 페이스북, 인스타그램 등 스크롤 기능이 있는 대부분의 애플리케이션은 최신 데이터가 가장 위에 나오고 스크롤은 아래로 내려가도록 구성되어 있다. 하지만 채팅 어플에서 메시지를 주고 받는 화면은 최신 데이터가 가장 아래에 나타나고 스크롤의 방향은 위로 올라가도록 화면이 구성된다.

FlatList 컴포넌트를 이용하여 아래부터 데이터를 렌더링하려면 어떻게 해야할까. FlatList 컴포넌트에는 inverted 속성이 있는데, 이 값에 따라 FlatList 컴포넌트를 뒤집은 것처럼 스크롤 방향이 변경된다..

```
JS ChannelList.js JS Channel.js X JS firebase.js
src > screens > JS Channel.js > [🔍] Channel
57
58   return(
59     <Container>
60       <FlatList
61         keyExtractor={item => item['id']}
62         data={messages}
63         renderItem={({item})=>(
64           <Text style={{fontSize: 24}} >{item.text}</Text>
65         )}
66         inverted={true}
67       />
68     <Input
69       label = "message"
```




화면에서 상단에 나타나던 메시지가 FlatList 컴포넌트의 inverted 값을 true로 설정하면 하단부터 나타나는것을 확인할 수 있다.

메시지를 보낸 사람에 따라 메시지의 렌더링 위치가 달라진다. 본인이 보낸 메시지는 오른쪽에 이미지 없이 나타나고, 다른 사람이 보낸 메시지는 이미지 및 사용자 이름과 함께 왼쪽에 나타나는 것이 우리가 흔히 사용하는 채팅 어플의 모습이다. 애플리케이션에 따라 다르지만, 사용자 사진의 경우 연속된 메시지에서는 한 번만 렌더링 하는 경우도 있다.

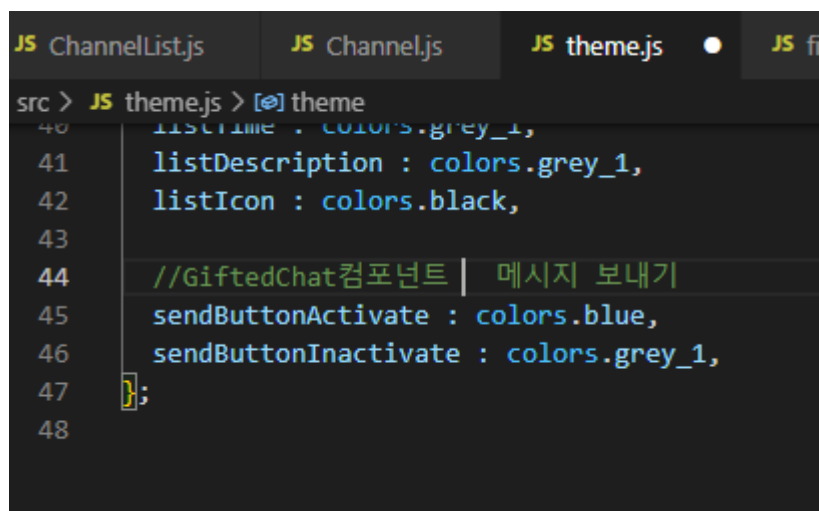
우리가 직접 다양한 컴포넌트를 생성해서 언급된 화면의 모습을 만들 수도 있지만 많은 시간과 노력이 필요하다. 그래서 이번엔 언급된 내용뿐 아니라 채팅 화면에서 사용할 수 있는 기능을 다양하게 제공하는 react-native-gifted-chat 라이브러리를 사용해서 화면을 구성하겠다.

아래 명령어로 라이브러리 설치를 진행하자

- `npm install react-native-gifted-chat (- -force)`

라이브러리의 GiftedChat 컴포넌트는 다양한 설정이 가능하도록 많은 속성을 제공한다. 입력된 내용을 설정된 사용자의 정보 및 자동으로 생성된 id와 함께 전달하는 기능뿐 아니라, 전송 버튼을 수정하는 기능이나 스크롤의 위치에 따라 스크롤 위치를 변경하는 버튼을 렌더링할 수도 있다.

먼저 GiftedChat 컴포넌트를 이용해 채널 화면을 수정하면서 사용할 색을 theme.js파일에 정의하자.



```
JS ChannelList.js JS Channel.js JS theme.js JS fir
src > JS theme.js > [x] theme
40 listTime : colors.grey_1,
41 listDescription : colors.grey_1,
42 listIcon : colors.black,
43
44 //GiftedChat컴포넌트 | 메시지 보내기
45 sendButtonActivate : colors.blue,
46 sendButtonInactivate : colors.grey_1,
47 };
48
```

이제 정의된 색과 GiftedChat 컴포넌트를 이용해 채널 화면을 수정하자.

```
JS ChannelList.js JS Channel.js JS theme.js JS firebase.js X
src > utils > JS firebase.js > ...
81   return id;
82 };
83
84 export const createMessage = async ({ channelId, message }) => {
85   const docRef = doc(db, `channels/${channelId}/messages`, message._id);
86   await setDoc(docRef, { ...message, createdAt: Date.now() });
87 };
```

```
JS ChannelList.js JS Channel.js X JS theme.js JS firebase.js
src > screens > JS Channel.js > ...
1  import React, { useState, useEffect, useLayoutEffect, useContext } from 'react';
2  import styled, { ThemeContext } from 'styled-components/native';
3  import { Alert } from 'react-native';
4  import { GiftedChat, Send } from 'react-native-gifted-chat';
5  import { MaterialIcons } from '@expo/vector-icons';
6  import { createMessage, getCurrentUser, app } from '../utils/firebase';
7  import {
8    getFirestore,
9    collection,
10   onSnapshot,
11   query,
12   doc,
13   orderBy,
14 } from 'firebase/firestore';
15
```

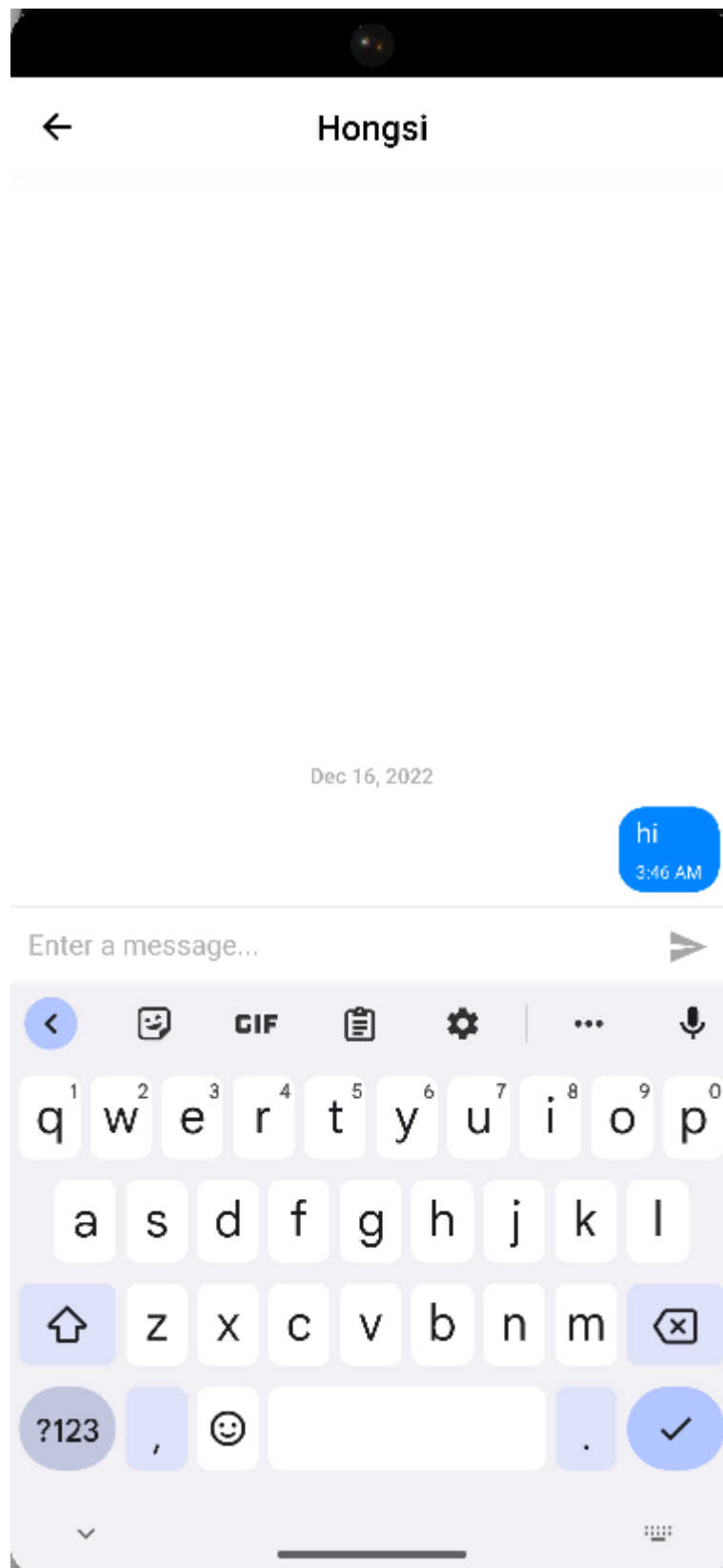
```
JS ChannelList.js X JS Channel.js X JS theme.js JS firebase.js
src > screens > JS Channel.js > ...
22 `;
23
24
25 const SendButton = props => {
26   const theme = useContext(ThemeContext);
27
28   return (
29     <Send
30       {...props}
31       disabled={!props.text}
32       containerStyle={{
33         width: 44,
34         height: 44,
35         alignItems: 'center',
36         justifyContent: 'center',
37         marginHorizontal: 4,
38       }}
39     >
40       <MaterialIcons
41         name="send"
42         size={24}
43         color={
44           props.text ? theme.sendButtonActivate : theme.sendButtonInactivate
45         }
46       />
47     </Send>
48   );
49 };
50
```

```
JS ChannelList.js JS Channel.js JS theme.js JS firebase.js
src > screens > JS Channel.js > Channel > _handleMessageSend
51
52
53 const Channel = ({ navigation, route }) => {
54   const theme = useContext(ThemeContext);
55   const { uid, name, photoUrl } = getCurrentUser();
56   const [messages, setMessages] = useState([]);
57
58   const db = getFirestore(app);
59   > useEffect(() => { ...
73   }, []);
74   > useEffect(() => { ...
76   }, []);
77
78   const _handleMessageSend = async messageList => {
79     const newMessage = messageList[0];
80     try {
81       await createMessage({ channelId: route.params.id, message: newMessage });
82     } catch (e) {
83       Alert.alert('Send Message Error', e.message);
84     }
85   };

```

```
87   return(
88     <Container>
89       <GiftedChat
90         listViewProps={{
91           style: { backgroundColor: theme.background },
92         }}
93         placeholder="Enter a message..."
94         messages={messages}
95         user={{ _id: uid, name, avatar: photoUrl }}
96         onSend={_handleMessageSend}
97         alwaysShowSend={true}
98         TextInputProps={{
99           autoCapitalize: 'none',
100           autoCorrect: false,
101           contentType: 'none', // iOS only
102           underlineColorAndroid: 'transparent', // Android only
103         }}
104         multiline={false}
105         renderUsernameOnMessage={true}
106         scrollToBottom={true}
107         renderSend={props => <SendButton {...props} />}
108       />
109     </Container>
110   )
111 }
112

```



기존에 화면을 구성하던 FlatList 컴포넌트와 Input 컴포넌트를 제거하고 GiftedChat 컴포넌트를 이용해서 화면을 구성했다.

GiftedChat 컴포넌트의 user에 다음과 같은 형태로 사용자의 정보를 입력해두면 onSend에 정의한 함수가 호출될 때 입력된 메시지와 사용자의 정보를 포함한 객체를 전달한다.

onSend에 작성된 함수는 파라미터로 다음과 같은 형태의 메시지 객체가 배열로 전달되며 전송 버튼을 클릭하면 호출한다.

메시지 객체에 자동으로 생성된 _id값은 UUID를 이용해 생성된다. 만약 _id 생성 방법을 변경하고 싶다면 messageIdGenerator에 _id를 생성하는 함수를 작성하면된다.

- uuid 라이브러리 : <https://github.com/uuidjs/uuid>

TextInput 컴포넌트는 여러 줄을 입력할 수 없게 설정하고, 전송 버튼을 작성된 SendButton 컴포넌트로 렌더링하도록 작성했다. SendButton 컴포넌트는 입력된 텍스트의 유무에 따라 다른 색으로 렌더링하도록 작성했다. 마지막으로 스크롤이 일정 수준 이상으로 올라가면 스크롤의 위치를 가장 아래로 이동시키는 버튼이 나타나도록 설정했다.

결과를 보면 앞에서 보낸 메시지에는 사용자 정보가 없으므로 현재 사용자와 다른 사용자라고 판단해서 왼쪽에 메시지가 렌더링되었다. GiftedChat 컴포넌트의 기능 중 메시지에 있는 createdAt 값을 이용해 날짜를 렌더링하는 기능도 확인할 수 있다. 마지막으로 스크롤 방향이 반대인 것과 메시지를 입력하면 전송 버튼의 색이 변경되는 것도 볼 수 있다.

메시지 생성 수정

데이터베이스에 메시지 문서를 생성할 때 파이어베이스에서 생성하는 id를 사용하도록 만들었다. 우리가 사용하는 GiftedChat 컴포넌트에서 생성한 고유의 값을 이용하도록 메시지 생성 함수를 수정하자.

채널 화면에서 onSend에 정의된 함수를 아래 코드처럼 수정한다.

```
hannelList.js JS Channel.js X JS theme.js JS firebase.js
> screens > JS Channel.js > [X] Channel
> |
|   useEffect(() => { ...
|   }, []);
|
|   const _handleMessageSend = async messageList => {
|     const newMessage = messageList[0];
|     try {
|       await createMessage({ channelId: route.params.id, message: newMessage });
|     } catch (e) {
|       Alert.alert('Send Message Error', e.message);
|     }
|   };
|
|   return(
|     <Container>
```

onSend에 정의된 함수에 파라미터로 전달되는 값을 이용해서 createMessage 함수를 호출하도록 작성했다. 전달되는 메시지에는 자동으로 생성된 _id와 입력된 메시지, 사용자의 정보를 담은 객체가 포함되어 있어 메시지를 전송한 사람의 정보를 전달하는 작업을 추가할 필요가 없다. 이제 firebase.js 파일에서 createMessage 함수를 전달되는 값에 맞게 변경하자.

```
JS ChannelList.js JS Channel.js JS theme.js JS firebase.js X
src > utils > JS firebase.js > ...
81 |   return id;
82 | };
83 |
84 | export const createMessage = async ({ channelId, message }) => {
85 |   const docRef = doc(db, `channels/${channelId}/messages`, message._id);
86 |   await setDoc(docRef, { ...message, createdAt: Date.now() });
87 | };
```