

[API reference](#)[Hooks](#)[useFocusEffect](#)**Version: 6.x**

# useFocusEffect

Sometimes we want to run side-effects when a screen is focused. A side effect may involve things like adding an event listener, fetching data, updating document title, etc. While this can be achieved using `focus` and `blur` events, it's not very ergonomic.

To make this easier, the library exports a `useFocusEffect` hook:

```
import { useFocusEffect } from '@react-navigation/native';

function Profile({ userId }) {
  const [user, setUser] = React.useState(null);

  useFocusEffect(
    React.useCallback(() => {
      const unsubscribe = API.subscribe(userId, user => setUser(user));

      return () => unsubscribe();
    }, [userId])
  );

  return <ProfileContent user={user} />;
}
```

Try this example on Snack [↗](#)

Note: To avoid the running the effect too often, it's important to wrap the callback in `useCallback` before passing it to `useFocusEffect` as shown in the example.

The `useFocusEffect` is analogous to React's `useEffect` hook. The only difference is that it only runs if the screen is currently focused.

The effect will run whenever the dependencies passed to `React.useCallback` change, i.e. it'll run on initial render (if the screen is focused) as well as on subsequent renders if the dependencies have changed. If you don't wrap your effect in `React.useCallback`, the effect will run every render if the screen is focused.

The cleanup function runs when the previous effect needs to be cleaned up, i.e. when dependencies change and a new effect is scheduled and when the screen unmounts or blurs.

## Running asynchronous effects

When running asynchronous effects such as fetching data from server, it's important to make sure that you cancel the request in the cleanup function (similar to `React.useEffect`). If you're using an API that doesn't provide a cancellation mechanism, make sure to ignore the state updates:

```
useFocusEffect(
  React.useCallback(() => {
    let isActive = true;

    const fetchUser = async () => {
      try {
        const user = await API.fetch({ userId });

        if (isActive) {
          setUser(user);
        }
      } catch (e) {
        // Handle error
      }
    };

    fetchUser();

    return () => {
      isActive = false;
    };
  }, [userId])
);
```

If you don't ignore the result, then you might end up with inconsistent data due to race conditions in your API calls.

## Delaying effect until transition finishes

The `useFocusEffect` hook runs the effect as soon as the screen comes into focus. This often means that if there is an animation for the screen change, it might not have finished yet.

React Navigation runs its animations in native thread, so it's not a problem in many cases. But if the effect updates the UI or renders something expensive, then it can affect the animation performance. In such cases, we can use `InteractionManager` to defer our work until the animations or gestures have finished:

```
useFocusEffect(  
  React.useCallback(() => {  
    const task = InteractionManager.runAfterInteractions(() => {  
      // Expensive task  
    });  
  
    return () => task.cancel();  
  }, [])  
);
```

## How is `useFocusEffect` different from adding a listener for `focus` event

The `focus` event fires when a screen comes into focus. Since it's an event, your listener won't be called if the screen was already focused when you subscribed to the event. This also doesn't provide a way to perform a cleanup function when the screen becomes unfocused. You can subscribe to the `blur` event and handle it manually, but it can get messy. You will usually need to handle `componentDidMount` and `componentWillUnmount` as well in addition to these events, which complicates it even more.

The `useFocusEffect` allows you to run an effect on focus and clean it up when the screen becomes unfocused. It also handles cleanup on unmount. It re-runs the effect when dependencies change, so you don't need to worry about stale values in your listener.

## When to use `focus` and `blur` events instead

Like `useEffect`, a cleanup function can be returned from the effect in `useFocusEffect`. The cleanup function is intended to cleanup the effect - e.g. abort an asynchronous task, unsubscribe

from an event listener, etc. It's not intended to be used to do something on `blur`.

For example, **don't do the following**:

```
useFocusEffect(  
  React.useCallback(() => {  
    return () => {  
      // Do something that should run on blur  
    }  
  }, [])  
);
```

The cleanup function runs whenever the effect needs to cleanup, i.e. on `blur`, unmount, dependency change etc. It's not a good place to update the state or do something that should happen on `blur`. You should use listen to the `blur` event instead:

```
React.useEffect(() => {  
  const unsubscribe = navigation.addListener('blur', () => {  
    // Do something when the screen blurs  
  });  
  
  return unsubscribe;  
}, [navigation]);
```

Similarly, if you want to do something when the screen receives focus (e.g. track screen focus) and it doesn't need cleanup or need to be re-run on dependency changes, then you should use the `focus` event instead:

## Using with class component

You can make a component for your effect and use it in your class component:

```
function FetchUserData({ userId, onUpdate }) {  
  useFocusEffect(  
    React.useCallback(() => {  
      const unsubscribe = API.subscribe(userId, onUpdate);  
  
      return () => unsubscribe();  
    }, [userId, onUpdate])  
  );  
}
```


```
    }, [userId, onUpdate])
  );

  return null;
}

// ...

class Profile extends React.Component {
  _handleUpdate = user => {
    // Do something with user object
  };

  render() {
    return (
      <>
        <FetchUserData
          userId={this.props.userId}
          onUpdate={this._handleUpdate}
        />
        { /* rest of your code */ }
      </>
    );
  }
}
```

 [Edit this page](#)