# Advanced event handling

Let's say that Dinner And A Movie want us to create a new component, a "people list" component where they can render a list of employees from their database, or customers who like a particular dish or like a particular film. It'll be your task to create that component. But we have a starter for you, a Person component and a Person type.

1. Look in your starters for a folder called people_starters. Move the Person.tsx file into your components folder and the Person.ts file into your types folder. Feel free to look through them to see what's happening.

We also need a bunch of people to test with.

2. Point your browser at https://randomuser.me. They provide an API that allows requesting of any number of fake but realistic looking people. In fact, look at some sample data by making a GET request to https://randomuser.me/api?results=10.

## Prepping for the PeopleList

You want to create a bare-bones structure first.

3. Create the new component called PeopleList. For now, return a big text:

```
<h1>People List</h1>
```

4. Don't add it to the NavBar yet because we're just writing a proof of concept for our Dinner And A Movie clients. But do add a Route for PeopleList in App.tsx.

5. Run and test. Make sure you can see your PeopleList by putting "http://localhost:5173/people" into the address bar. Adjust the port if needed.

## Fetching a few people

6. Create a state variable called "people" with a useState. It is a list of People objects.

7. In the component's return, iterate (map) through your people, and render one Person component per person in the list.

8. Run and test. You should see no difference yet because there are no people in the list. Let's get some people, shall we?

## Fetching a few people

9. Write a function in the component called fetchPeople. It should ...
    - fetch 10 people from RandomUser.me.
    - Add them to the existing list of people in state.

10. Add a button whose onClick method invokes fetchPeople.

11. Run and test. Click the button. If you've done this right, you should now see 10 people cards on your screen. Click it again and you have 20 people. Another click will show you 30 people.

12. Bonus! Your people cards are probably stacked up in the y-direction. Wrap them in a container with display set to 'flex' and flex-wrap set to 'wrap'. So that they line up in a grid-like way.

# Fetching the first 10 on load

13. Create a useEffect() that runs one time on page load only. Call fetchPeople() from there.

14. If you've done this right, then 10 people appear immediately and the others are added on button click.

# Adding a scroll event listener

Our UX experts have determined that users don't want to have to click the button to fetch more people. They claim that a better paradigm is to have the component recognize when the user has scrolled to near the end of the list. They want us to implement infinite scroll.

The trick is to listen for the scroll event. On every scroll event, we should check to see if we're nearing the end of the list and if so, fetch another batch of people. We do this infinitely until the user exits the component.

So we want to tap into the onScroll event on ... what, exactly? The root node? If so, the root node of what? This component? If you look closely you'll see that React doesn't support an onScroll event, at least in the way you'd expect it to be supported.

We clearly need a different approach.

Let's escape React's control and create a low-level event listener for the window.

```
window.addEventListener('scroll', () => doSomething());
```

But there's a problem with that. If you wire it up in the body of the component, it gets wired up over and over and over, every time the component is rendered. You may end up with 100 scroll listeners that all do the same thing and never go away.

Let's use useEffect to wire up the event on load and then clean it up on unload.

15. In your existing useEffect() that runs one time on load, replace the call to fetchPeople with a scroll event listener. It should call a function called handleScroll. For now make handleScroll merely console.log() ...
    - The page height: document.documentElement.scrollHeight
    - The viewport height: window.innerHeight and
    - The scroll position: window.scrollY.

16. Run and test. If your scrolling console.logs a bunch of numbers, you're doing it right. Using these numbers, you can calculate how close your user is to the bottom of the screen.
    - pixelsFromBottom = pageHeight - (viewportHeight + scrollPosition)

# Cleaning up

We need to clean up after ourselves by deleting the scroll listener we added or it will literally never go away.

17. Change your useEffect to remove the scroll event listener when the component is disposed of. (Hint: Put the console.log()ging logic in a function called handleScroll and do this:

```
window.removeEventListener('scroll', handleScroll);
```

18. Edit the logic in handleScroll to say that when we scroll near the bottom of the scene, console.log(). If we're not near the bottom of the scene, do nothing. You can decide what "near" means to you. A couple hundred pixels should do the trick.

19. Run and test.

# Doing the actual fetch

20. Edit handleScroll. Make it call fetchPeople instead of console.log()ging.

When you've got this component infinitely scrolling people, you can be finished. Way to go!

# Bonus!! Solving a few challenges

You probably already saw that the scroll event fires many times per second. You may find that many fetch requests are dispatched and returned before the window can redraw itself.

21. Don't send a fetch if the last one hasn't return yet. (Hint: add a useRef variable called 'loading').

22. Add a "Fetching more people message at the bottom in case the user ever scrolls faster than we can load.