React Practice

App Component

```
export default function App() {
  async function getAdvice() {
    const response = await fetch("https://api.adviceslip.com/advice");
    const data = await response.json();
    console.log(data);
  }

  return (
    <div>
      <h1>Hello World!</h1>
      <button onClick={getAdvice}>Get Advice!</button>
    </div>
  );
}
```

→JSX

```
import { useEffect, useState } from "react";

export default function App() {
  /* useState is a function that returns an array in react
   and here we are destructuring that array 1st position of the
   array is value of the state, the 2nd value is a setter function that
   we can use to update the piece of state */
  const [advice, setAdvice] = useState("");
  const [count, setCount] = useState(0);

  async function getAdvice() {
    const res = await fetch("https://api.adviceslip.com/advice");
    const data = await res.json();
    setAdvice(data.slip.advice);
    /* Now in this getAdvice function we can use setAdvice function to
    update the state, whenever the piece of state is updated
    user interface will also be updated */
    // take current count add 1 and that will become the new count
    setCount((c) => c + 1);
  }
  // Generate very first piece of advice when loaded
  /* useEffect takes two arguments 1st a function that
  we want to execute at the beginning when the component loads and 2nd
  argument is dependency array */
```

```jsx
  useEffect(function () {
    getAdvice();
  }, []);
  return (
    <div>
      <h1>{advice}</h1>
      <button onClick={getAdvice}>Get Advice!</button>
      {/* include this component like it is another html element */}
      {/* Pass count as a prop to Message. Props are like parameters
      to function. We call the prop count and pass in the prop
      value */}
      <Message count={count} />
    </div>
  );
}


// In react we divide user interfaces into components
// components are reusable pieces of code
// components are used to render UI
// Name of all components should start with capital letter(convention)
// Now accept the props object as a parameter. In this props object count is
now a property
function Message(props) {
  // update count dynamically
  return (
    <p>
      You have read <strong>{props.count}</strong> pieces of advice
    </p>
  );
}
```

JSX Template: Babel transpiles JSX code by transforming it into regular
JavaScript code. JSX (JavaScript XML) is a syntax extension for JavaScript that
allows developers to write HTML-like code within JavaScript. Babel is a
toolchain that is primarily used to convert ECMAScript 2015+ (ES6+) code into a
backwards-compatible version of JavaScript that can be run in older browsers or
environments.

When Babel encounters JSX code, it transforms it into function calls that
create React elements. Here's a simplified explanation of how Babel transpiles
JSX:

1. **Parsing**: Babel first parses the JSX syntax into an Abstract Syntax Tree
(AST), which is a structured representation of the code.

2. **Transformation**: Babel then transforms the JSX AST into equivalent JavaScript code. This transformation involves replacing JSX elements with `React.createElement()` function calls.

3. **Generation**: Finally, Babel generates the transpiled JavaScript code from the transformed AST.

For example, given the following JSX code:

```jsx
const element = <h1>Hello, world!</h1>;
```

Babel transpiles it into the following JavaScript code:

```javascript
const element = React.createElement("h1", null, "Hello, world!");
```

In this transpiled code, `React.createElement()` is used to create a React element representing the `<h1>` element with the text content "Hello, world!".

It's important to note that Babel itself doesn't understand JSX syntax natively; instead, it relies on plugins like `@babel/preset-react` to handle JSX transformation. This preset includes the necessary plugins to transform JSX into JavaScript that can be understood by browsers or other JavaScript environments.

App.js:

```
import "./App.css";
function App() {
  const title = "Welcome to the new blog!";
  const likes = 50;
  // can output: number, string, and array
  // can't output: boolean and objects
  const link = "https://www.google.com/";

  const ninja = { name: "Yoshi", age: 30 };
  // returning dynamic values
  return (
    <div className="App">
      <div className="content">
        <h1>{title}</h1>
        <p>Liked {likes} times</p>
        {/* <p>{ninja}</p> */}
```

```
      <p>{`${ninja.name} is a ${ninja.age} years old ninja!`}</p>
      {/* bunch different element in arrays together and bunch them as string
*/}
      <p>{[1, 25, 686]}</p>
      {/* use dynamic values as attribute values in element tags */}
      <p>{Math.random() * 20}</p>
      <a rel="noreferrer" target="_blank" href={link}>
        Link to google!
      </a>
    </div>
  </div>
  );
}

export default App;
```

Components contain template which makes up the html along with js logic.
Components return jsx and In background a transpiler named babel converts all
this jsx template into regular JavaScript code when we save the file. class is
a reserved keyword in js show we can't use in jsx.

Components return jsx and exported to use elsewhere.

React converts out (whatever data type we give it to it) into strings before it
renders it in the browser.

 can output: number, string, and array
 can't output: boolean and objects

Lesson 6 (Adding Styles):

React styles are not bound to a single component, it will be applied to all
components in the browser at that time (inspect to see), cause react takes all
these styles adds them to head of the webpage. check out styles tag inside head
element.

The styles are not scoped into single component, we can use css modules or
styled components to scope styles into components.

We can create a global index.css file to implement all our styles.

Inline styling in react:

```
        {/* Inline styling in jsx, the first {} denote dynamic value and inside
{} denote an object inside its key-value pair, key = css property, value = css
property value (string) */}
        <a
          href="/create"
          style={{
            color: "#fff",
            backgroundColor: "#f1356d",
            borderRadius: "8px",
          }}
        >
          New Blog
        </a>
```

Lesson 7 (click events):

```
Home.js
-------
const Home = () => {
  const handleClick = (e) => {
    console.log("Hello Ninjas!", e);
  };
  const handleClickAgain = (name, e) => {
    console.log(`Hello ${name}!, ${e.target}`);
  };
  return (
    <div className="home">
      <h2>Homepage</h2>
      {/* passing function reference {handleClick} on clicking function will be
invoked, calling it handleClick() will invoke the function without clicking
*/}
      <button onClick={handleClick}>Click me!</button>
      {/*Avoid this handleClickAgain(name) as it will invoke the function
without clicking it, passing in argument to function: wrap inside anonymous
function '() => {
          handleClickAgain("Mario");
        }', e.g. handleClick is similar like () => { console.log(Hello
Ninjas!)} as we are not invoking the function with () we are referencing it, on
clicking the anonymous function is invoked then handleClickAgain("Mario") with
"Mario" argument, we can also remove curly braces ({}) cause it's one line, in
jsx {} indicates dynamic value */}
      <button onClick={(e) => handleClickAgain("Mario", e)}>
        Click me again!
      </button>
```

```
      {/* Event object/parameter we automatically get access to it when an
event occurs. As the first parameter of the function (referenced function), In
the second case: anonymous function gets access to event object, now will pass
that in handleClickAgain("Mario") function as 2nd argument */}
    </div>
  );
};


export default Home;
```

Lesson 8(useState hook):

State of a component means data being used on that component in a particular
time.
It could be an array of values, booleans, string, objects or any other data
that our component uses.

We created some variables and used them in our template before (refer to React
Practice.pdf App.js file) but if we need to change variable or data over time
or in reaction to some event (user clicking a button)

Home.js (Before using useState Hook):

```
// Destructure useState hook from react library, this grabs us the function
from the react library, now we can use it in our component
import { useState } from "react";

const Home = () => {
  let name = "Mario";
  const handleClick = () => {
    // Updating the value will not reflect in the template
    name = "Luigi";
    /* name itself changed to luigi but it doesn't get updated in the template
(view in browser console) cause the variable we have created is not reactive
(let name = "Mario") means react doesn't watch it for changes, when its value
changes it doesn't trigger react to re-render the template with the new value
inside it (name variable in this case) and we continue to see the old value
(Mario) in the browser, to make this work we need to make the value reactive so
that when it changes react detects that and it re-renders the template with the
new value (where we output it in the template) and updated value is visible to
the browser to implement this we use hook which is called useState(). Hook in
react is a special type of function that does a certain job, it generally
starts with the word use. useState hook provides us a way to make reactive
```

```
value, also provides a way to change that value whenever we want. So to use
useState hook we need to import it  */
    console.log(name);
  };

  // const handleClickAgain = (name, e) => {
  //   console.log(`Hello ${name}!, ${e.target}`);
  // };

  return (
    <div className="home">
      <h2>Homepage</h2>

      <p>{name}</p>
      {/* onclicking button change the value of name variable */}
      <button onClick={handleClick}>Click me!</button>
      {/* <button onClick={(e) => handleClickAgain("Mario", e)}>
        Click me again!
      </button> */}
    </div>
  );
};

export default Home;
-------------------------------------------------------------------------------

Home.js ( After using useState Hook ):
-------------------------------------------------------------------------------
// Destructure useState hook from react library, this grabs us the function
from the react library, now we can use it in our component
import { useState } from "react";

const Home = () => {
  // let name = "Mario";
  // Make a reactive value using useState, we give this useState a initial
value for example (Mario), we want to store this in some variable. We use array
destructuring to grab 2 values that this hook returns to us, first value is the
initial value (name), 2nd value is a function that we can use to change that
value most times it's called set[whatever the name of value to be changed]
  const [name, setName] = useState("Mario");
  const [age, setAge] = useState(23);
  const handleClick = () => {
    /* changing the state data using setName, this useState value is reactive
if we change it it's gonna change in the template as well, when we use this
function to change the value that triggers react to re-render the component
upon re-rendering it has the new value of name cause it's been updated. We can
```

use this hook (useState) as many times in our component for different values
other than name like it can be array, object, boolean etc. like 'const [age,
setAge] = useState(0)' The data type of state we are using doesn't matter */
    setName("Luigi");
    // update age to 30
    setAge(30);
  };

  return (
    <div className="home">
      <h2>Homepage</h2>
      {/* At first it will give us the initial value (Mario), to change this
value we can use the setName function */}
      <p>
        {name} is {age} years old.
      </p>
      <button onClick={handleClick}>Click me!</button>
    </div>
  );
};

/* Conclusion: When we need a reactive value something that might change at
some point we use the useState hook to do that we pass in an initial value and
we can output that value in the template and then we just call the set function
2nd value we get in the destructured array to update it and that triggers
re-render and the new value is going to be output to the browser in this
template, so this hook is very useful 😄 */

export default Home;
```

--------------------------------------------------------------------------------

**Lesson 9 (React Dev Tools):**
React Dev tools integrate with browser development tools and gives us extra
features that we can use on any website created with react. To extra tabs
components, and profiler is available. Components is more useful. This gives us
component diagram or component tree of our current app. Hovering over it
provides extra info about the component.

Important tabs in react dev tools:
1. Inspect the matching DOM element. (eye icon)
2. Log this component data to the console. (bug icon)

Change of state (data) under hooks tab can be viewed in home (by selecting it
from component tree) component by clicking the button (click me). Changes in
state in a component can be tracked here.

If we log the home component to the console we can see the hooks property with
array of objects and each object represents the piece of state we have with
properties name, value, id, isStateEditable etc.
--------------------------------------------------------------------------------

**Lesson 10 (Outputting Lists):**
Goal: Outputting list of blogs in our template.

Create some states to represent these blogs. We will be using useState hook
cause data might change at some point we might delete the blog, we need react
to update the DOM when that happens.

Home.js (lecture 10: Outputting List)
----------
```
import { useState } from "react";

const Home = () => {
  /* destructure the two values, initial value of this state is an array of
objects, each objects represent a blog with title, body, author, and id
property. This id is going to be used by react when we output this data, each
id needs to be unique for each one of the items/blogs. */
  const [blogs, setBlogs] = useState([
    { title: "My new website", body: "lorem ipsum...", author: "mario", id: 1
},
    { title: "Welcome party!", body: "lorem ipsum...", author: "yoshi", id: 2
},
    {
      title: "Web dev top tips",
      body: "lorem ipsum...",
      author: "mario",
      id: 3,
    },
  ]);
  return (
    <div className="home">
      {/* iterate/loop through blogs array using map method, for each iteration
as we iterate through this we get access to the item we are currently iterating
that is blog in this case, when we output list using map method each root
element in the template that we return must have a key property now this key
property is something that react uses to keep track of each item in the dom as
it outputs it, so if data changes at any point say we remove/add new items to
the array react can keep track of those items. So we must add a key attribute
to each item that we output, otherwise react can't distinguish between list
items in the DOM. This normally an id property for each item in the array (we
already have that in our array of objects) */}
      {blogs.map((blog) => (
```

```
      // what we want to output for each blog
      <div className="blog-preview" key={blog.id}>
        <h2>{blog.title}</h2>
        <p>Written by {blog.author}</p>
      </div>
    ))}
  </div>
  );
};
```

/* Summary: That is how we output a list of data in react. We have a list in
this case an array of object which is stored in useState the we map through the
data and we take each item into that as we map through it an we output a bit of
template for each one (in this case see comment what we want to output for each
blog) and each one has a key property which is id in our case but it could be
any unique property. Now some css is added for each blog template (view after
'blog previews / list' comment in index.css) */

export default Home;

Lesson 11 (Props):
If we are building a real blog we might have the list of blogs in various
places on our website it might be the homepage, search, category or tag page.
So several different areas may use the same logic where we cycling/iterating
through blogs and outputting a blog preview for each one. To implement that in
our project we'll be repeating the code (map function iteration) over and over
again in different components for different pages. Where we have pieces of
components or bits of templates that might be reused in different positions or
different places in the website we like to make that bit of template into its
own reusable component. e.g. if we make a component blog list then we could
drop this blog list component in any other components in the project. So if we
have category page later on we could just get the blog list component and drop
it in. To pass in different data into reusable component every time use it we
will do that in form of props. E.g. in the homepage we might list all the blogs
and show a preview for all the blogs starting from the latest one but on a
search page or search component, we might only show the blogs that match the
search term so the data is going to be different the structure is the same (of
map function iteration BlogList component) but the blogs that we are going to
use (state array of objects/blogs) is going to be different, so we can pass in
data into these external components as well in the form of props. An external
component of BlogList will contain all the logic/template(jsx) of map iteration
and listing blog preview.

Home.js

```

```jsx
import { useState } from "react";
import BlogList from "./BlogList";

const Home = () => {
  const [blogs, setBlogs] = useState([
    { title: "My new website", body: "lorem ipsum...", author: "mario", id: 1
},
    { title: "Welcome party!", body: "lorem ipsum...", author: "yoshi", id: 2
},
    {
      title: "Web dev top tips",
      body: "lorem ipsum...",
      author: "mario",
      id: 3,
    },
  ]);
  return (
    <div className="home">
      {/* {} - Dynamic value */}
      <BlogList blogs={blogs} title="All Blogs!" />
    </div>
  );
};
/* Conclusion: That's how we can make a component take in props data and then
use that data inside that component. It makes the Bloglist component more
reusable and it does, we can now use this BlogList component anywhere in our
application whether in home component or in different page component later on
*/
export default Home;
------------------------------------------------------------------------------
BlogList.js
-------------------
// const BlogList = (props) => {
const BlogList = ({ blogs, title }) => {
  /* storing blog property of props object in blogs, we are passing in
properties to props object and grabbing this different properties from this
props object and storing these in this variables now an easier way to do this
is destructuring ('{blogs, title}') as we want blogs and title from props
object */
  //   const blogs = props.blogs;
  //   const title = props.title;
  //   const { blogs, title } = props;
  //   console.log(props, blogs);
  return (
    /*  In the BlogList component we are trying to map through the blogs data
but this component has no idea of what blogs is. The blogs data is not defined
```

in this component. We can't just use any data in another where it's defined (Home component in this case) right here in home component because it can't reach that. There is 2 ways to fix this: 1st option is to redeclare all this data into BlogList component instead of home (data - ' const [blogs, setBlogs] = useState([

```
    { title: "My new website", body: "lorem ipsum...", author: "mario", id: 1
},
    { title: "Welcome party!", body: "lorem ipsum...", author: "yoshi", id: 2
},
    {
      title: "Web dev top tips",
      body: "lorem ipsum...",
      author: "mario",
      id: 3,
    },
  ]);'
```

the 2nd option is to use props whereby we pass this data from this home component into the BlogList here the 2nd option props will be used for 3 reasons 1st: It's gonna make our BlogList component more reusable and we'll see exactly how later on. 2nd: It allows us to still use this data in home component later on (cause we are not removing it) if we need it in future cause the data is still gonna be declared here in home component. 3rd: It allows to learn how to use props 😁. Props are a way to pass data from one component(a parent component) into a child component. In this case Home is parent component and BlogList is child component, we will be passing the blogs data into the BlogList component to do that we will make a property name on <BlogList/> tag. We can call it whatever we want it's blogs in this case. So, <BlogList blogs={blogs} /> now this is being passed into BlogList component (check home component code) as a prop (blogs={blogs} - this is a prop). We need to receive it here in BlogList component we get access to an argument inside this function/component called props. Now this blogs property will be in props object. Any props that we send through (like blogs={blogs}) into a component will be attached to this props object which we automatically get as an argument in the component ('const BlogList = (props)=>{}') and we can access them like props.blogs. We can pass in multiple props if we want to like we can pass in title="All Blogs" with a string value */

```
    <div className="blog-list">
      <h2>{title}</h2>
      {/* {props.blogs.map((blog) => ( */}
      {blogs.map((blog) => (
        // what we want to output for each blog
        <div className="blog-preview" key={blog.id}>
          <h2>{blog.title}</h2>
          <p>Written by {blog.author}</p>
        </div>
      ))}
    </div>
```

```
  );
};

export default BlogList;
```

Lesson 12 (Reusing components):

We took all this logic (map iteration) from home component and externalized it
into BlogList component and we passed props into that component the data which
it uses and externalizing all this logic into different component, makes that
code more reusable, we can reuse this component in different places in our
application where we need it and we can pass different data to it each time.

Home.js

```
import { useState } from "react";
import BlogList from "./BlogList";

const Home = () => {
  const [blogs, setBlogs] = useState([
    { title: "My new website", body: "lorem ipsum...", author: "mario", id: 1
},
    { title: "Welcome party!", body: "lorem ipsum...", author: "yoshi", id: 2
},
    {
      title: "Web dev top tips",
      body: "lorem ipsum...",
      author: "mario",
      id: 3,
    },
  ]);
  return (
    <div className="home">
      <BlogList blogs={blogs} title="All Blogs!" />
      {/* pass in filtered data from blogs array, we will take out the blogs
which doesn't have a author of mario, pass in title of 'Mario's Blogs' which
was 'All Blogs!' previously, filter method fires a callBack function for each
item in the array now if we return true for that item it keeps it in the array
and if we return false it filters it out of the array and it returns a new
array with the items except filtered out ones and we are passing this that data
(filtered out array) as a prop, so this BlogList component can be reused and we
can reuse it with different data which makes it really useful for doing things
like a search page whereby the title matches the search term for example */}
      <BlogList
```

```
      blogs={blogs.filter((blog) => blog.author === "mario")}
      title="Mario's Blogs!"
    />
  </div>
  );
};

export default Home;
-------------------------------------------------------------------
BlogList.js
-----------
const BlogList = ({ blogs, title }) => {
  return (
    <div className="blog-list">
      <h2>{title}</h2>

      {blogs.map((blog) => (
        <div className="blog-preview" key={blog.id}>
          <h2>{blog.title}</h2>
          <p>Written by {blog.author}</p>
        </div>
      ))}
    </div>
  );
};

export default BlogList;
------------------------------------------------------------------------
```

Next Lesson: Updating the state by deleting items from data array (state of array of objects/blogs)

Lesson 13 (Functions as Props Delete an item):

Allow users to delete blogs by clicking on the button. We need a button inside the BlogList component for each blog that we output doing that below the author {blog.author}.

```
BlogList.js
-----------
const BlogList = ({ blogs, title, handleDelete }) => {
  /* function to delete blog using id: here we want to delete the blog with
that id from blog data, now the data is initialize in home component useState
(this is where the state is), we don't want to directly edit the blogs prop
(blogs prop - that is passed in in this component from Home component) (we
```

shouldn't do that) instead we need to use setBlogs method inside Home component to the update the state (change the state after deleting an item (blog)) that's what we need to do, it's not good to defined handleDelete function in here (in BlogList component) instead we want it to be defined in the home component, so we can interact with the data directly and then we can pass in this data (returned data from handleDelete function, the data after deleting) through as prop, so we can create a prop called handleDelete in Home component and set it equal to handleDelete function e.g. handleDelete={handleDelete} and the in BlogList we can accept this function as a prop in BlogList component and we are using it in the onClick event see below ⬇️, So we are invoking this function which is defined in the parent home component and inside home component we can use setBlogs function to update the state and we can remove the blog with this id (check handleDelte function in Home component) */

```
  return (
    <div className="blog-list">
      <h2>{title}</h2>
      {blogs.map((blog) => (
        <div className="blog-preview" key={blog.id}>
          <h2>{blog.title}</h2>
          <p>Written by {blog.author}</p>
          {/* button with click event handler (onClick), invoke an anonymous
function while clicking it (Note: We pass in function expression (body of
function or function name)), we'll pass in the id of the blog we want to delete
so that we can find it in the array and delete it, we have access to the blog
and the id property is in the blog, so we are passing that (blog.id) into
handleDelete function so we know in here which blog to delete, this click event
invokes the handleDelete function defined in Home component */}
          <button onClick={() => handleDelete(blog.id)}>Delete blog!</button>
        </div>
      ))}
    </div>
  );
};

export default BlogList;
```

Home.js

```
import { useState } from "react";
import BlogList from "./BlogList";

const Home = () => {
  const [blogs, setBlogs] = useState([
```

```javascript
    { title: "My new website", body: "lorem ipsum...", author: "mario", id: 1
},
    { title: "Welcome party!", body: "lorem ipsum...", author: "yoshi", id: 2
},
    {
      title: "Web dev top tips",
      body: "lorem ipsum...",
      author: "mario",
      id: 3,
    },
  ]);

  const handleDelete = (id) => {
    /* store the new array temporarily in newBlogs, the filter method doesn't
change the original (blogs) array it doesn't mutate (change) it, It returns a
new filtered array, blog id that doesn't match this passed id will remain in
the array id that is not matched is removed, the new array is stored in
newBlogs */
    const newBlogs = blogs.filter((blog) => blog.id !== id);
    /* we will use setBlogs to set new value (filtered array of objects/blogs
(newBlogs)) for blogs, see the defined array of objects up 👆 in the useState
that is previous (initial) value and we will update the state by setting the
value of blogs to newBlogs, fingers crossed 🤞, upon refreshing the modified
newBlogs state will not persist the state will set back to initial state as it
re rendering, re-running the code, that's how we are passing functions
(handleDelete) into other components (BlogList) as props from parent
component(Home) */
    setBlogs(newBlogs);
  };

  return (
    <div className="home">
      {/* we can pass in this data (returned data from handleDelete function,
the
      data after deleting) through as prop, so we can create a prop called
      handleDelete in Home component and set it equal to handleDelete function
      e.g. handleDelete={handleDelete} */}
      <BlogList blogs={blogs} title="All Blogs!" handleDelete={handleDelete} />
    </div>
  );
};

/* conclusion: handleDelete function is defined where the original initial data
is in the home component we will not be modifying blogs prop instead we will
pass this handleDelete function as prop to BlogList component and use it in
there in the onClick method by invoking it with the id of the blog to be
```

deleted this handleDelete function defined in Home component since then invoked
and uses setBlogs(newBlogs) to change value of blogs to newBlogs filtered array
upon refreshing the code reruns and the state(data) of blogs is set back to
initial value */

export default Home;

Lesson 14 (useEffect Hook):

We have seen a useState hook that is used to create some state for the
component. But there are many other hooks that we can use in React. One such is
useEffect - this hook runs a function at every render of the components,
remember the component renders initially when it first loads and the rendering
also happens when the state changes it re renders the dom so we can update that
state in the browser, so this useEffect hook is a way to run code on every
render and that can be useful for many different reasons which we're going to
see later on, now focus on how can we use it.

First thing to do import it from react - (import { useState, useEffect } from
"react")

Home.js

```
import { useState, useEffect } from "react";
import BlogList from "./BlogList";

const Home = () => {
  const [blogs, setBlogs] = useState([
    { title: "My new website", body: "lorem ipsum...", author: "mario", id: 1
},
    { title: "Welcome party!", body: "lorem ipsum...", author: "yoshi", id: 2
},
    {
      title: "Web dev top tips",
      body: "lorem ipsum...",
      author: "mario",
      id: 3,
    },
  ]);

  const handleDelete = (id) => {
    const newBlogs = blogs.filter((blog) => blog.id !== id);
    setBlogs(newBlogs);
  };
```

```
   /* use useEffect hook - we don't store it inside a constant it doesn't return
anything all we need to do is pass as an argument a function, this function the
function that's going to run every time there is a re-render so once initially
when the component first loads but thereafter anytime the data changes,
normally in this function inside useEffect we could do something like fetch
data or communicate with some kind of authentication service and those things
are known as side effects in react, but for now we are doing simple
console.log, if we see in browser console we can observe useEffect ran on
refresh and when we delete a blog(changing the data (state) (re-rendering)) so
useEffect runs on every render. We can also access the state inside useEffect,
so if we want to output the blogs we can do 'console.log(blogs)' inside
useEffect. If we inspect the browser console we can see the blogs data on every
render as useEffect runs, Need to be careful about changing the state inside
useEffect because because we could end up in a loop of continuous renders (e.g.
setting state inside useEffect 'setBlogs(newBlogs)') 😅. In this scenario,
Initially the component renders to the DOM  which will trigger useEffect
function to run that would then update the state and the state(data) would
change and that would trigger a re-render on that re-render again that triggers
this function in useEffect to run and this goes on again and again creating a
endless loop, there are ways to fix it which we will see later */

  useEffect(() => {
    console.log("useEffect ran");
    console.log(blogs);
  });

  return (
    <div className="home">
      <BlogList blogs={blogs} title="All Blogs!" handleDelete={handleDelete} />
    </div>
  );
};

/* Conclusion: This useEffect hook is really really useful for running any kind
of code that we need to run at every render. It can be used for things like
fetching data we're gonna see that later. Next up we gonna look at dependencies
of useEffect */
export default Home;
-----------------------------------------------------------------------------
```

Lesson 15 (useEffect Dependencies):

useEffect hook the function inside it fires after every render, that happens
once initially when the component first loads but thereafter every time the
state changes and we re-render the template. But we don't want to run a
function after every single render rather maybe only after certain renders to

do that we can use dependency array, this is an array that we can pass into useEffect hook as a second argument like this:

```
useEffect(() => {
  console.log("useEffect ran");
  console.log(blogs);
}, []);
    👆 (2nd argument)
```

Passing an empty array - this ensures that useEffect hook runs only after the first initial render, thereafter if the state (data) changes it won't run the function again. It only runs it once. This is useful if we want to only run the function once after the first render.

Now we can also add actual dependencies to this array, meaning any state (data) values that should trigger the useEffect function to run when they change. To demonstrate we are creating another piece of state.

Home.js

```
import { useState, useEffect } from "react";
import BlogList from "./BlogList";

const Home = () => {
  const [blogs, setBlogs] = useState([
    { title: "My new website", body: "lorem ipsum...", author: "mario", id: 1
},
    { title: "Welcome party!", body: "lorem ipsum...", author: "yoshi", id: 2
},
    {
      title: "Web dev top tips",
      body: "lorem ipsum...",
      author: "mario",
      id: 3,
    },
  ]);

  const [name, setName] = useState("Mario");

  const handleDelete = (id) => {
    const newBlogs = blogs.filter((blog) => blog.id !== id);
    setBlogs(newBlogs);
  };

  /* We want to run this useEffect at the beginning when the component first
renders but also whenever a certain value changes (in this case name meaning if
```

this state changes -   `const [name, setName] = useState("Mario")` so the useEffect function will run only when the name state changes, given name will become the dependency we add into the 2nd argument `[name]` - now useEffect is going to watch this value and if it changes it will run the function. Now on the first render useEffect still runs displayed console.log(name) output mario on browser console. If if we delete the blogs it will not run cause it's only watching for changes in name state not in blogs state cause blogs is not in the dependency array, but if we change the name it does run the function (see the console) cause name is in the dependency array & when it changes the function inside useEffect is ran). Now one thing to notice: after changing the name state to luigi, if we click the change name button again it will not run again. Cause although it is using setName("Luigi") function to change the state it's not actually changing the value anymore cause it's already luigi at this point (we already clicked the button changing the state to luigi) so state is not changing and we are not triggering that re-render and therefore the function inside useEffect (useEffect) in general is not running */

```
  useEffect(() => {
    console.log("useEffect ran");
    // console.log(blogs);
    console.log(name);
  }, [name]);

  return (
    <div className="home">
      <BlogList blogs={blogs} title="All Blogs!" handleDelete={handleDelete} />
      {/* this onClick event will invoke an anonymous function which then
invokes the setName(function that changes the state - data) */}
      <button onClick={() => setName("Luigi")}>Change name</button>
      {/* Outputting, the name in a paragraph */}
      <p>{name}</p>
    </div>
  );
};
```

/* Conclusion: That's how we can use dependencies this dependency array is the 2nd argument to useEffect to control when this useEffect function runs */

export default Home;

Lesson 16(Using Json Server):

Fetching Data using useEffect - it is good place to fetch data in a component cause we know it runs the function (function inside useEffect sent as 1st argument) when the component first renders initially, and that's generally when

we want to go and fetch some data and then we can use that data in our
application instead of the data that we already have in the blogs state:

```
`const [blogs, setBlogs] = useState([
    { title: "My new website", body: "lorem ipsum...", author: "mario", id: 1
},
    { title: "Welcome party!", body: "lorem ipsum...", author: "yoshi", id: 2
},
    {
      title: "Web dev top tips",
      body: "lorem ipsum...",
      author: "mario",
      id: 3,
    },
  ])`
```

Cause typically in an web app we will not have hard coded data like this,
instead it will probably come from a database using an api endpoint (rest api).
We are gonna using json server which will allow us to build a fake rest api
just using a json file that we can use to test this out.

1st step is to create a json file which is going to act as our database, this
will reside in data folder of root directory of dojo-blog. in `data/db.json`

[db.json is one property called blogs with an array of two other objects]

```
db.json
-------
{
"blogs": [
    {
      "title": "My First Blog",
      "body": "Why do we use it?\nIt is a long established fact that a reader
will be distracted ",
      "author": "mario",
      "id": 1
    },
    {
      "title": "Opening Party",
      "body": "Why do we use it?\nIt is a long established fact that a reader
will be distracted ",
      "author": "yoshi",
      "id": 2
    }
  ]
}
```

so each object is a blog with title, body, author, and id, we have two
objects/blogs in total and when we're using json server each top level property
is considered a resource so we just have one top level property blogs (which
contains the array with 2 objects) so it sees that as resource and creates
endpoints for us to interact with this resource so we can do things like delete
items from it, add items to it, edit items, get the items etc so that is
db.json file in a nutshell.
   // comments are not allowed in json

Now we'll use json server package to watch this file (db.json) and wrap it with
some endpoints. So there is 2 options here either 1. Install json server
package locally into this project and then use it or 2. Use npx like we did to
create-react-app to run the code from web and it will still watch our file
right here `db.json`, that's is what we are gonna do. Open up different
terminal rather than that's running our local development
server(localhost:3000).

We need to install json-server with 2 flags: (watch and port)
`npx json-server --watch data/db.json --port 8000`

watch followed by the path of the file to watch.
And the port number where json-server will run.

After running it (the command above) is going to watch db.json and will wrap it
with some api endpoints.

It picked up that we have a blogs resource: http://localhost:8000/blogs
So if we want to send a get request to get all of the blogs we would use this
endpoint: http://localhost:8000/blogs - we can view this url in browser and see
the data typically we're not gonna be using the browser to get that data
instead we're gonna be using a fetch request inside our component using these
different endpoints right here. Now at the moment we can see this:
http://localhost:8000/blogs endpoint but it also provides us with other
endpoints. The endpoints we are gonna be using (provided by json-server to us)
is shown in the JSON server API endpoint image in lecture summary. This is not
all of the endpoints, just the ones we will be using. The use of these
endpoints is also provided in the image. The first one we already tested in the
browser (refer to the image) - http://localhost:8000/blogs.

/blogs/{id} - http://localhost:8000/blogs/1 (1 is the id)

The Image:
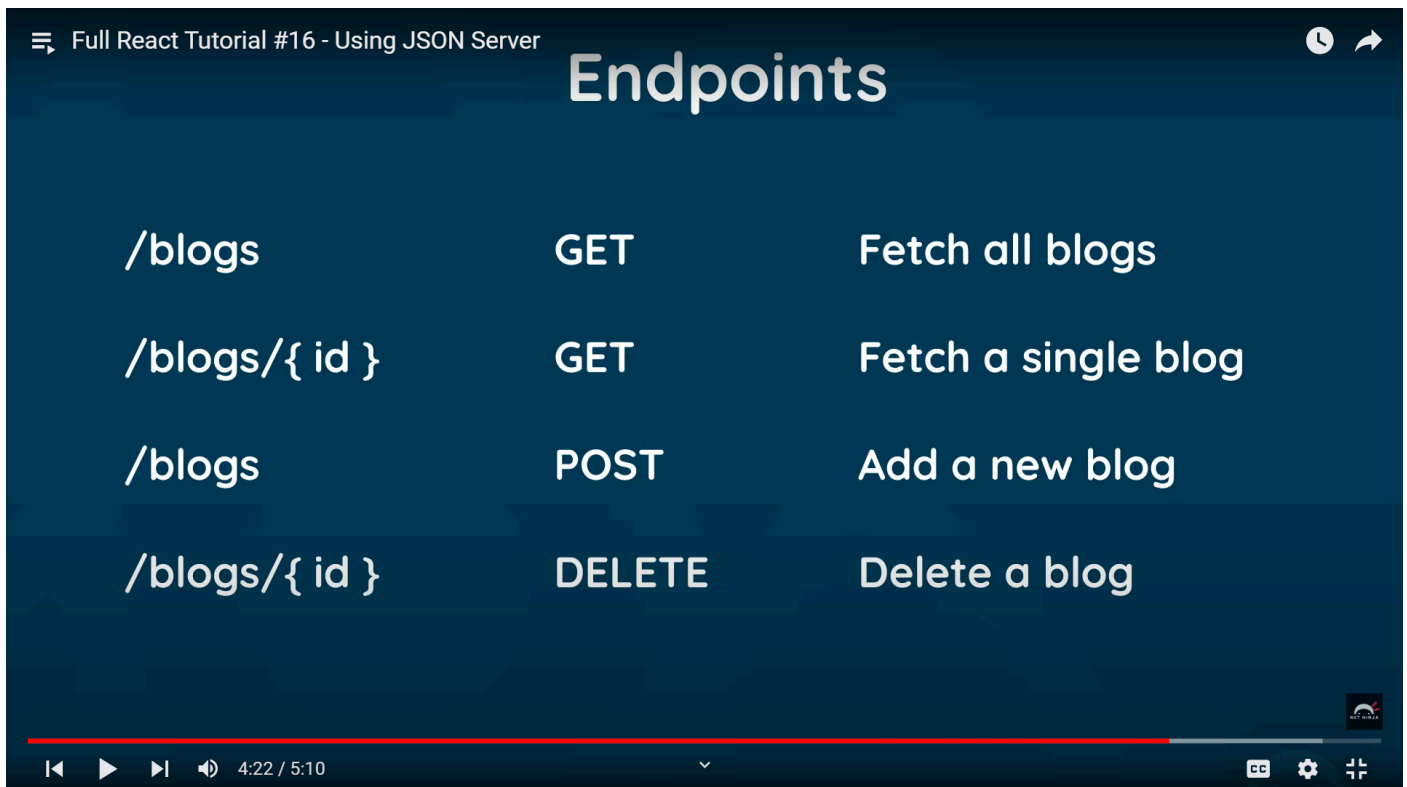
Lesson 17 (fetching data using useEffect):

Fetch data inside useEffect hook - it's going to run when the component first
renders that's is the only time it's going to run cause we passed an `[]` empty
dependency array to useEffect as 2nd argument.

We are not gonna use initial data hardcoded data for blogs state instead we
will set the initial value as `null`. Upon successfully fetching the data from
db.json file we are going to update the `blogs` state (data) using `setBlogs`
with the data we get back.

Let's make that fetch request, fetch('endpoint') = format of fetch

Home.js

```
import { useState, useEffect } from "react";
import BlogList from "./BlogList";

const Home = () => {
  const [blogs, setBlogs] = useState(null);

  // const handleDelete = (id) => {
  //   const newBlogs = blogs.filter((blog) => blog.id !== id);
  //   setBlogs(newBlogs);
```

```
  // };

  useEffect(() => {
    /* get request to `http://localhost:8000/blogs` this returns to us a
promise, we can't use async we can't use await inside this to implement that we
need to externalize a function with all of this logic inside it and make that
asynchronous. But here we going to use `then` method, so this will fire a
function once this promise has resolved so once we have the data back, first of
all we get a response object, this response object is actually not the data is
just a response object and in order to get the data we have to do something
with the response object and this is when we just use the fetch api
(`http://localhost:8000/blogs`), So we need to use `res.json()` to pass the
json into javascript object */
    fetch("http://localhost:8000/blogs")
      .then((res) => {
        /* and this passes the json into a javascript object, we need to return
this value and when we return this it returns another promise cause res.json()
is also asynchronous it takes some time to do. so we need chain another then
method which is going to fire a function once `res.json` is complete and this
function (function inside then method that is chained 2nd time) takes in as
parameter the actual data that `res.json` gets us. So use the res (response)
object use the json method on that (res.json) and chain another then method
whereby we get the data. So this data is going to be whatever in here in
db.json in javascript format or rather a javascript array (the array starting
from the top level resource `blogs`). We are logging the data inspect browser,
now we are going to update initial null value given to the blogs state using
setBlogs, if we put this setBlogs function inside useEffect it is not going to
cause an infinite loop cause we have put the empty dependency array ensuring it
only runs after first initial render not whenever the data changes */
        return res.json();
      })
      .then((data) => {
        console.log(data);
        setBlogs(data);
      });
  }, []);

  return (
    <div className="home">
      {/* Upon uncommenting the following line there might still be a error,
here we are passing blogs as a prop to the BlogList component and then inside
the BlogList the error is occuring in the map method, where we trying to map
through the blogs that we pass through now the blogs should be the data that we
have seen in the console `console.log('data') in 2nd chained then method. So
why it is not working? It's not working cause it takes a fraction of time to
get that data and initially the value of blogs is null and we're passing in
```

null (in below line as prop to BlogList `blogs={blogs}`) when the app first
loads in the browser so in the BlogList component blogs (blog property
destructured from props object and stored in blogs `blogs = props.blogs`) is
null and we are trying to use map method on null at the very start once we get
the data it should work fine but to begin with until we have that data we get
the error. So how do we combat this? So we don't want to output the below line
until we have a value for blogs. So we are going to short circuit && operator
by wrapping the whole line in {}which denotes javascript [&& ShortCircuiting:
When first value is true the and operator will automatically return the second
operand. No matter what that is. When first value is false (here null is a
falsy value) the and operator will automatically return first value and doesn't
check the second value. That is short circuiting. This acts little bit like if
statement]. This is called conditional templating in react. So if the the left
operand (here that is null and evaluates to false) is false it doesn't even
bother checking the right operand (it short circuits), the <BlogList/> is not
outputted here. If the blogs left operand evaluates as true, this moves to the
right operand and evaluates <BlogList/> by evaluating this it outputs it to the
screen. So the right operand is only ever going to be output if the left
operand is true and that's generally how we can conditionally output parts of
template🔥(I think it means one template or the other) we're going to see more
of this later. So that's all */}
        {blogs && <BlogList blogs={blogs} title="All Blogs!" />}
      </div>
    );
};

/* Conclusion: So we are fetching the data as soon as the component first
renders at that moment in time once we have the data we update the state right
here `setBlogs(data)` once we update the state it has a value it outputs this
`<BlogList/>` and this passes through that value `blogs={blogs}` then we cycle
through them in BlogList component and render them to DOM */

/* Now we are going to remove the buttons to delete the blog because if we are
deleting the blog it's just from local data and yes it'll work because we're
storing this into local data in the state `in blogs state. Also notice we're
updating blogs state upon deleting using setBlogs(newBlogs) in handleDelete
function that runs on onClick event` but ultimately we are going to make delete
request to db.json file so let's get rid of handleDelete functionality cause we
don't need that anymore from Home component and in BlogList component, that is
passed function as prop to BlogList component. So now it's gone */

/* Next up we're going to see more about conditional templates (using logical
operators) and how to output a loading message as we're trying to fetch the
data */
export default Home;

```
BlogList.js
-----------
/* Removing the handleDelete function that was passed as prop from home
component to BlogList component, also the button that is used to invoke it */

const BlogList = ({ blogs, title }) => {
  return (
    <div className="blog-list">
      <h2>{title}</h2>
      {blogs.map((blog) => (
        <div className="blog-preview" key={blog.id}>
          <h2>{blog.title}</h2>
          <p>Written by {blog.author}</p>
          {/* <button onClick={() => handleDelete(blog.id)}>Delete
blog!</button> */}
        </div>
      ))}
    </div>
  );
};

export default BlogList;
```

Lesson 18 (Conditional Loading Message):

Currently in our app we render the BlogList `{blogs && <BlogList blogs={blogs}
title="All Blogs!" />}` once we have blogs data and until then we don't render
it. Now it would be nice to create a loading message while the data is being
fetched so that user knows something is loading if it takes time to do, now our
fetch is very quick because we're just making a fetch locally to our computer.
But most times fetch will be to another server over the internet and slower in
which case the user will see that loading message while we fetch the data.

To do this, we're going to create an additional piece of state inside the home
component and this is going to be called isPending and the function to change
it's value that is setIsPending with initial value set to true:

```
  const [isPending, setIsPending] = useState("true");
```

Now we will implement another conditional template ->

```
Home.js
-------
import { useState, useEffect } from "react";
import BlogList from "./BlogList";
```

```jsx
const Home = () => {
  const [blogs, setBlogs] = useState(null);
  const [isPending, setIsPending] = useState("true");

  useEffect(() => {
    /* Since it happens too fast we're gonna add 1 sec/1000 millisec delay to
it, so that we can show the loading message. This setTimeout function (the
anonymous function inside it) will fire after 1 sec then we can perform the
fetching operation, so that just makes fetch a little more realistic in terms
of the amount of time it takes to get the data. Don't use setTimeout for
real-world apps 😅 cause we're just forcing our user to wait another 1 second
when they don't need to. This is just to kind of simulate the idea of the
request taking little bit of time so it makes it a bit more realistic */
    setTimeout(() => {
      fetch("http://localhost:8000/blogs")
        .then((res) => {
          return res.json();
        })
        .then((data) => {
          console.log(data);
          setBlogs(data);
          // Setting isPending to false once we have the data, therefore
updating the state
          setIsPending(false);
        });
    }, 1000);
  }, []);

  return (
    <div className="home">
      {/* only when isPending is true we want to show a loading message. Now to
begin with isPending is true it is true as it's initial value set to true so
user will see it. This shows it, but we want to show it only when the fetch is
going on and We don't want to show it once we have the data. So once we have
the data in 2nd chained .then method available to us we can set isPending to
false. See Up ☝️ . Upon refreshing we can see the loading h2 for split second.
To see it better we can use chrome dev tools and set the throttling to
mid-tier-mobile. */}
      {isPending && <h2>Loading......</h2>}
      {blogs && <BlogList blogs={blogs} title="All Blogs!" />}
    </div>
  );
};

/* Conclusion: That's how we can conditionally output a loading message */
export default Home;
```

Next up we're gonna look at fetching errors🔥

Lesson 19 (Handling Fetch Errors):

One more thing to do in home component is to handle any kind of error when we
try to make the fetch. That could be an error which is sent back from the
server or it could be connection error where we can't even connect to the
server in these cases we wouldn't be getting the data back and we need to let
the user know that there is some kind of error. So the first thing to do, is
add a catch block (chain this method to 2nd chained then method)

Home.ejs

```
import { useState, useEffect } from "react";
import BlogList from "./BlogList";

const Home = () => {
  const [blogs, setBlogs] = useState(null);
  const [isPending, setIsPending] = useState("true");
  // Creating state to show the error in the browser, to begin with this state
will have the initial value of null
  const [error, setError] = useState(null);

  useEffect(() => {
    setTimeout(() => {
      fetch("http://localhost:8000/blogs")
        .then((res) => {
          /* Read 2nd: take the response object `res` and check the `ok`
property on the response object and in fact before we do that we are going to
comment the if block and log the response object (res). Now check the logged
response object where we get `ok` property. The value of this `ok` property
will be true if everything is fine and we get the data back. If we don't get
the data back for example if we send a request to an endpoint that is faulty or
that doesn't exist then value of `ok` property will be false, so what we want
to do is check is the response okay or rather is it not okay because if this
`ok` property is false the we want to throw an error. So we'll uncomment the if
block and we will check not if the response is okay but if it's not okay
(!false = true, condition will be true and the if block will execute) now at
this point in this if block we want to throw an error, cause it means there is
an error coming back from the server so we'll throw it right here like this
(this is how we throw an error): throw Error('Our own error message'). So when
we throw an error in this fetch it catches it in the catch block/method. So if
it's not ok it throws the error and we catch that error with the message
attached to it. So we should see that message in `console.log(err.message)`.
Now at the moment it's not gonna fire that message because that response is
```

okay (ok property is true) but if we change our endpoint to something that doesn't exist like add another `s` see the fetch argument ☝️. So the endpoint will become faulty it's going to make request to the server and the server will send an error back so ` if (!res.ok) {
            throw Error("Could not fetch the data for that resource");
        }` block will be fired and then we will throw our own error and we'll catch this error in catch block. So it shows in the console `Could not fetch the data for that resource`. So now we're catching the errors what we would like to do to store the error in some kind of state so that we could output it to the browser   */
        // console.log(res);
        if (!res.ok) {
          throw Error("Could not fetch the data for that resource");
        }
        return res.json();
      })
      .then((data) => {
        console.log(data);
        setBlogs(data);
        // Setting isPending to false once we have the data, therefore updating the state
        setIsPending(false);
        setError(null);
      })
      .catch((err) => {
        /* Read 1st: This catch block catches any kind of network error, and it will fire a function (that anonymous function with err (we can name this parameter anything we want) parameter). We'll log this error to console. So this will catch any kind of network error, so that's if we can't even connect to server, now we can simulate this kind of error if we open the terminal and terminate (ctrl+c) the json-server. Now we shouldn't be able to connect to the server cause it's not running, and we should catch the error in catch block an will log that to the console. That is a connection error but what if there is another type of error for example imagine our request reaches the server but the server sends an error back maybe if the endpoint that we've tried to fetch from doesn't exist or if the request is denied, in that case this catch block over here doesn't automatically catch those errors when we use the fetch api cause it's still reaching the server and the server is still sending a response(res in this case) object back to us it's just that the response doesn't contain the data and it will contain a different status now in this case we need to check that response object when we get it back, in the first then block we're going to do a if check look up ☝️ */
        /* Read 3rd: So When we catch the error over here instead of logging it to the console, what we can do to setError to update the state by passing in the error message (err.message property) so whatever message is in the error that could be the error message we are throwing `Could not fetch the data for

that resource` or the network error message that we get when we are not able to
connect to server, that we saw terminating the json server */

```
          // console.log(err.message);

          // set isPending message to false when having an error
          setIsPending(false);
          setError(err.message);
        });
    }, 1000);
  }, []);

  return (
    <div className="home">
      {/* Now we can output this error message in our template, if we have a
value for error state (data), again using conditional templating or conditional
rendering by using && operator short circuiting. So we have only have a value
for error this output will be visible. Otherwise it will have the initial falsy
value of `null`. So if we run it the browser we can see it working, however we
still se the Loading...... message cause isPending state is true as we not
updating it as we haven't got the data back because of the error (look the 2nd
chained then method). We don't want to see that Loading...... if we have an
error because it's not actually still loading so what we could do is if we get
an error we also want to set this state (isPending) to be false so it doesn't
show that loading message. So we are gonna do that as well. See up ☝️ . No we
will set the error to null inside 2nd chained then method if we get data
because if we try to make subsequent request by fetching the data again (after
already getting the data) at any point we want to get rid of the error message
if it's successful so the previous error(any error connection or faulty
endpoint fetch error etc.) is resolved then there is no point showing the
previous error in the template therefore error state is going to be changed to
null. So we are gonna do setError(null) by setting the state to null thereby
updating the state */}
      {error && <div>{error}</div>}
      {isPending && <div>Loading......</div>}
      {blogs && <BlogList blogs={blogs} title="All Blogs!" />}
    </div>
  );
};

export default Home;
```

Lesson 20 (Making a custom hook):

We put together all this logic inside the useEffect hook to update all of these
all of these state properties to output the data or a loading message or an

error if there is one and everything is working fine and it's fine to be left like this but what if we wanna do same kind of thing in another component in the future where we fetch some data we create state for the data itself the error and isPending property etc. We'd essentially have to rewrite all of this code in that component again and that's not very efficient, (no re usability as we need to rewrite all this in another component) It's not easy to manage especially if we're using this code in few different places in our application. So it would be good if we can make use of all of this code again in different components so make it a bit more reusable so we don't have to continually write it out again and again for example what we could do is we could rip it all out of this component and put in it's own javascript file and then we just import that into this file to use it and we could also import that into any other component that might need to use the same logic in the future and this way we're only writing and maintaining the code in one place (in it's own javascript file) and not in several different places over different components that need to fetch data. So when we do something like this by externalizing the logic into its own file we're creating something called a custom hook in react so a bit like useState and useEffect have their own specific functionality as hooks we'd be creating a custom hook with specific ability to fetch data, So in order to make this custom hook the first thing we're going to do is make a new file inside the source `src` folder called `useFetch.js`.

Inside this we'll place all the functionality to fetch data exactly as we did exactly in the home component. But first of all we need to create a function to put the code in and this function will be the hook itself.

useFetch.js

/* custom hooks in react needs to start with the word use otherwise it won't work, Now grab the whole useEffect thing in here now paste it inside the hook/function (useFetch - this function called a hook cause it starts with the word use). Now we need to do some tweaks first of all we need to import the useEffect hook and we also need to register all of the states as well because currently the state is in home component, but we're not setting the state in home component anymore we're setting them here in useFetch. So we're grabbing all the states and bringing them here from home component. We'll also import useState and useEffect in order to use them. We'll export this function/hook (Component functions starts with capital letter this function is a hook not an component) to use it in other components and we'll call the blogs state data cause we want to make this reusable so in the future if we're using this hook fetch data it might be to fetch a different type of resource for example tags or categories so it makes no sense to call it blogs all the time. So change state name to data in the function to update this state to setData. Notice the parameter inside the first chained then method is also called data but it will not affect us cause it's local version inside this function (the anonymous function). So it doesn't matter if the names clash. Last thing to do is to

return some values from useFetch function. So we know hooks(useState, useEffect) return some values, we can do the same thing in our custom hooks, so at the bottom useFetch function we're gonna return some values */

```
import { useState, useEffect } from "react";
const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [isPending, setIsPending] = useState("true");
  // Creating state to show the error in the browser, to begin with this state
will have the initial value of null
  const [error, setError] = useState(null);

  useEffect(() => {
    setTimeout(() => {
      fetch(url)
        .then((res) => {
          // console.log(res);
          if (!res.ok) {
            throw Error("Could not fetch the data for that resource");
          }
          return res.json();
        })
        .then((data) => {
          console.log(data);
          setData(data);
          // Setting isPending to false once we have the data, therefore
updating the state
          setIsPending(false);
          setError(null);
        })
        .catch((err) => {
          // console.log(err.message);
          // set isPending message to false when having an error
          setIsPending(false);
          setError(err.message);
        });
    }, 1000);
  }, [url]);

  /* return an object and place 3 values inside this object, the return value
```

can be whatever we like it could be an array like useState(useState returns an array we destructure that array) or it could be a string or boolean in our case it's an object with three properties they are data,isPending, and error cause when we use this useFetch hook inside another file/component in future we'll be able to grab this properties from the hook that's what we want to get. One more thing we want to do is to pass in the api endpoint (url) into the function rather than hardcode it in `fetch("http://localhost:8000/blogs")` cause if

we're using this useFetch hook in another file or in different component at
some point it might not always be to the same endpoint (url). So we'll use a
parameter of url in useFetch function and inside fetch as well `fetch(url)`.
Now we need to pass an argument to this useFetch url parameter whenever we want
to call it, that also means that we're gonna place url as dependency to the
useEffect dependency array `[url]` and that means whenever the url changes it's
going to rerun this function to get the data for that endpoint. (assuming we're
going to create state variable of url and setUrl function to update it in other
component - Not sure about this 😐). Now we're going to import this useFetch
function in our Home component and then use it */
  return { data, isPending, error };
};
// export it to use it in other components
export default useFetch;

Home.js
----
import { useState, useEffect } from "react";
import BlogList from "./BlogList";
// import useFetch required
import useFetch from "./useFetch";

const Home = () => {
  /* destructure the three properties we get back returned from useFetch
function, we could have returned array in useFetch and destructure it here, but
returning object is beneficial cause then order of these properties doesn't
matter as long as the destructured variable names and property names are
same***, and we can just grab isPending without getting other returned
properties if that's what we want. Now the initial value to this hook we need
to pass in the endpoint as argument to useFetch(url) parameter. So that's the
resource we're trying to fetch */
  /* data: blogs - this means grab the `data` but call it blogs in this
context, so now when we pass blogs in we're just passing in data `blogs =
data`. Let's save and see this if it works ☝🏼 */
  const {
    data: blogs,
    isPending,
    error,
  } = useFetch("http://localhost:8000/blogs");

  return (
    <div className="home">
      {error && <div>{error}</div>}
      {isPending && <div>Loading......</div>}
      {/* Changing the blogs state to data */}
      {blogs && <BlogList blogs={blogs} title="All Blogs!" />}

```
      </div>
    );
};
// *** Need to verify it in browser console

/* So that's how we can make a custom hook. So now this is more reusable now we
don't have to redo all of this logic in every component that needs to make a
fetch all we need to do is one line of code destructure and pass the endpoint
in (doesn't matter what that is ) It's going to try and fetch that data and
bring it back to us we get a loading state (isPending) and also an error if
there is one which we can use in that component in general. This results into
neat and tidy components that are easier to read */
export default Home;
```

Lesson 21 (The React Router):

So far, our application has only single page, we don't navigate around to other
pages, we just have single home page. Most websites we create going to have
probably more than one page, so we need a way to introduce multiple different
pages or routes in our react application and the way we do this in react is
with the react router but before we talk about how that works let us know how a
typical multi-page website works.

Typical website not using react handling multiple pages: Start in the browser
by typing a url in the address bar and hitting enter that sends a request to
that server for that address and the server handles it. The server is generally
going to send back full html page which we then view in a browser now if a user
was to click on a link on that website to another page like the contact page it
then sends a brand new request to the server and then the server responds by
sending back a new contact html page and we view that in the browser and the
cycle would continue over and over as we click other page links on the website.
So we're constantly making request to the server for new pages.

Websites using react handling multiple pages:
 Now react applications don't work like this. They delegate all the routing and
changing of page content to the browser only and it starts the same way we make
an initial request in the browser the server then responds to that by sending
back the html page to the browser but it also sends back our compiled react
javascript files which controls our react application. So from this point on,
react and the react router can take full control of the application. So
initially, the html page we get back is virtually empty and then react injects
the content dynamically using the components that we create if we then decide
to click on a link to a new page the react router is going to step in and
intercept that request to stop it from going to the server and instead it's
going to look at the new page request and inject the required content on the
screen. For example clicking on a contact link the react router will tell react

to inject the contact component in the browser, if we were to click in a about link it would tell react to inject the about component and so forth. So this is generally the way react router works. We assign a top-level component for each route or page and that component is dynamically injected into the browser when we visit that route. Now this whole process, means that we're making less request to the server and the whole website therefore feels faster and slicker. Now we know from a bird's eye perspective how it works. Let's set it up in our code:

We need to install react router package cause it's not part of the core react library. So on a new terminal execute(@5 - version 5, this version is used in this tutorial):
`npm i react-router-dom@5`

Now look into `package.json` if it's there, so remember when we install something it goes into the `node_modules` folder.

Set up routing for our application: 1st thing to do go to the root component that is App.js, we need to import few things from react-router package in here.

App.js

```
// / - forward slash \ - back slash
// don't need to provide .js at filename
import Navbar from "./Navbar";
import Home from "./Home";
/* lesson21: (Read the comments top to bottom) To get started with routing we need to destructure few things, 1st thing we need is BrowserRouter, we're gonna say `BrowserRouter as Router` that means we can use the BrowserRouter that we're importing using this name `Router` inside this file. We also want to import something called the `route` component and also the `switch` component, we'll see what they do as we go forward. So now we need to surround our entire application with Router component and that means we can use the router in the entire application all components that are nested inside this app component will have access to the router. */
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

function App() {
  // const title = "Welcome to the new blog!";

  return (
    <Router>
      {/* lesson21: this div will be surrounded by Router component,
```
<Router><div>.....</div></Router> so that's the first step, The next step is to decide where we want our page content to go when we go to different pages, we

```jsx
want go inside the div with className content. So we're gonna delete the home
component  */}
      <div className="App">
        {/* self closing custom components, we can do <Navbar></Navbar> as well
*/}
        <Navbar />
        <div className="content">
          {/* lesson21: We're gonna delete the home component and we're gonna
replace it with the `Switch` component. This Switch component makes sure that
only one route shows at any one time. More about it will be discussed later.
Just know for now, all of our routes go inside this `Switch` component. So now
we need to setup our individual routes, so what we do is we create a route for
each page that we have using this `Route` component, now we only have one page
so we're just going to place one route inside this `Switch` component but later
on we're gonna have other pages and more routes inside here as well. So let's
do our route for the home page. */}
          {/* <Home /> */}
          <Switch>
            {/* lesson21: So the Route component like so with the closing tag
and add a property to the route component which is going to be `path`, this
`path` is basically the route, so for the home page it would just be '/' the
root path. For example for contact page it would be '/contact' for about
'/about' and so on, basically this is the path after the root of our website.
e.g. if our root is junayed.com it can be the path after that root path of our
URL like junayed.com/about, '/' is the path for home page and we need to nest
the `Home` component inside inside this `Route` that we want to show when a
user visits this route `path="/"` so we wanna show the `Home` component. So
basically it's saying I want to show the Home component right here inside the
div with className content when we visit '/' path. Notice: the Navbar
`<Navbar/>` component always going to show because it's not inside this switch
statement. Meaning this component is here for every single path or route but
the content on the `Switch` is going to change depending on the route as we
build up more routes. So let's save and preview🔥. Now nothing really should
change because we're still using just the same home page but at least it does
work just forward slash in the address bar and we get the home page 🚀. Next up
we're going to add another route and will discuss more about this `Switch`
component we've used */}
            <Route path="/">
              <Home />
            </Route>
          </Switch>
        </div>
      </div>
    </Router>
  );
}
```

```
export default App;
```

Lesson 22 (Exact Match Routes):

Let's add another route to our app to do this we need another page component.
And this is going to for a web form to add a new blog so we're gonna call this
component `Create`. So we'll create that inside the source folder named
`Create.js` and we'll create a stateless functional component (sfc) in here
named Create.

Create.js

```
const Create = () => {
  // This template will be very simple to begin with, Now we will show this
component App.js (App Component)
  return (
    <div className="create">
      <h2>Add a new blog:</h2>
    </div>
  );
};


export default Create;
```

App.js

```
import Navbar from "./Navbar";
import Home from "./Home";

import { BrowserRouter as Router, Route, Switch } from "react-router-dom";
// import create component
import Create from "./Create";

function App() {
  // const title = "Welcome to the new blog!";

  return (
    <Router>
      <div className="App">
        <Navbar />
        <div className="content">
          {/* <Home /> */}
          <Switch>
```

```
        <Route exact path="/">
          <Home />
        </Route>
```
{/* To add another route add multiple different components inside the `Switch` this time the path is going to be create '/create' and it's not going to render the component it's going to render the `Create` component. Also import the 'Create' component on top. let's see if this works, and the results are probably not gonna be what we expected. If we execute the create route we get the `Home` component rendered that doesn't make sense, so why it's showing us the homepage/`Home` component when we've gone to '/create' which the path registered in our `Route` `path="/create"` that's do with the way react looks at route paths and matches them against our routes that we create right here. In this case, it sees the route we go to as '/create' what we type in the address bar and then it looks through our routes and it tries to match that against with one of our paths right here now it matches against the first one it finds to be a correct match and it stops there. So it goes from top to bottom and it looks at this first path '/' first of all and it says well yes this is a match that's strange cause it's not a match but in the eyes of the react router it is kind of a match and that's because this thing right here this '/' (forward slash thing), this route is inside '/create' (this forward slash exists in that '/create' route) and this logic might seen strange but that's the way it works. So '/create' is considered a match for '/' cause it contains that forward slash `/`. Imagine if we have a route for '/c' this would match the for '/create' as well cause create route contains '/c'. So how do we get around this, cause we don't want to show the `Home` component if we go to '/create' (that makes no sense). All we have to do is say well look I don't want you to match this if it's included in the route we only want we want you to match this if it's exactly the route so all we do is add on the exact prop in '/' route like this see up☝️. So that's it. So now it's saying only match this if it's exactly the url we go to not just inside the url. So it's to do with order as well if we put the '/' path after '/create' it will work just fine. So we'll add more routes and pages in the same way in the future.*/}

{/* `Switch` component surrounds all routes and it is there to make sure only one route component shows in the browser at any one time. So when a request is made to a route react will look at that request and then go from top to bottom inside the switch and try to match against a route and it's going to stop at the first match it finds for that route and render that component inside the route. Now if the `Switch` component doesn't surround all of these routes it would carry on and render other matches as well meaning we could end up more than one page showing at one time. So it's always a good idea to surround our routes with `Switch` component. Now there is still a problem with this setup when we click on links to go to different pages it's still sending a request to the server (notice it reloads) for each click so for example. So if we click on New blog btw that goes to '/create' it's still making a request to the server now we said that react router should intercept those requests and handle it in the browser instead well it does but instead of using anchor tags

```
we have to a special react router link instead and we'll see that next up
[cliff hanger 😐]    */}
              <Route path="/create">
                <Create />
              </Route>
          </Switch>
        </div>
      </div>
    </Router>
  );
}


export default App;
```

Lesson 23 (Router Links):

We have set up multiple different routes now for '/' and '/create' but when we
click on these links to those different routes it's still currently sending a
fresh request to the server for a new page, we get the same html page (that is
blank) back each time and then react is injecting the correct content but we
know react router can intercept those requests and just handle the content
changes in the browser instead and it can do but to do it we need to use a
special link tag so let's go to the `Navbar` component and try to use this,
first thing we need to do is to import this link tag from react.

Navbar.js

```
// Component is just a function which returns jsx template and the exported at
the bottom of file
// sfc (stateless functional component) press tab
/* lesson 23: Importing link tag from react, destructure the link component
from react-router-dom to handle new page requests in the browser (react router
intercepts these request to the server ) thus making no request to the server
for new pages.  */
import { Link } from "react-router-dom";

const Navbar = () => {
  return (
    <nav className="navbar">
      <h1>The Dojo Blog</h1>
      <div className="links">
        {/* We are currently using anchor(a) tags but if we want react to
handle the routing only in the browser and to intercept those requests for new
pages we need to replace this with `Link` instead `<Link></Link>` now a `Link`
```

component doesn't have href like an anchor tag instead it uses `to`. So now
we're saying we want to link to '/' and '/create'. So at the end of the day
when react renders all this to the browser we still will see anchor tags and we
can see that in browser inspecting elements, so when clicked on (the navbar
buttons that go to below routes 👇) they still try to send a request to the
server for a new page however built into these link tags is a special
functionality and that is for react router to have the ability to prevent that
request to the server and then instead it just looks at the url or the path
where we are going to and then it matches that against one of our routes
defined in App.js (App/Root component) and it tries to inject whatever content
that we need because we don't really need the server to resend back the html
page doesn't make much sense. Now if we see it in the browser it is much
quicker than sending a fresh request every time we want a new page to the
server, but one thing to notice (lesson 23, time: 2:40 - 3:30): Let us go to
'/' homepage and refresh, from Home to New blog if we go quickly we then get an
error and it says warning can't perform a react state update on an unmounted
component, now this is to do with our custom fetch hook which is still running
in the home page ('Home' component). It's still running in the background even
though we've gone to a new page over here and it's trying to update the state
of our `Home` component after it's been unmounted from the DOM and replaced
with this `Create` component. Now we are gonna see how to combat this error
using a cleanup function in our `useEffect` hook next */}
        <Link to="/">Home</Link>
        <Link to="/create">New Blog</Link>
      </div>
    </nav>
  );
};

export default Navbar;

Lesson 24 (useEffect cleanup):

So now whenever we land on our home page, this fetch for data starts using
using our custom hook (useFetch) and once it completes it tries to update the
state in the home component and we see that effect right here because when the
state changes we output that to the DOM that data however if while the data is
still trying to be fetched while it's still loading we then go to another page
before the fetch completes we get an error. So from homepage `/` we are going
to new blog `/create` now the fetch is still happening on the background once
we've switched to add a new blog '/create' and therefore when the fetch is
complete it still tries to update that state in the `Home` component but the
home component isn't in the browser any more (cause we're in the `/create'
route that renders `Create` component) that's why we get this error because

it's saying we can't perform a react state update on an unmounted (not in the
browser currently) component. The unmounted component is the `Home` component.
So we need a way of stopping the fetch when the component using it unmounts. So
if we go home then quickly to new blog we want to stop that fetch from carrying
on now to do this we'll be using a combination of the cleanup function in a
useEffect hook and something called an abort controller in javascript. So first
of all we're going to do is go to the useFetch hook that we have right here

useFetch.js

```
import { useState, useEffect } from "react";
const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [isPending, setIsPending] = useState("true");
  // Creating state to show the error in the browser, to begin with this state
will have the initial value of null
  const [error, setError] = useState(null);
  /* lesson 24 (1st read): We wanna run a cleanup function when the component
that uses this hook `useFetch` and this `useEffect` right here unmounts. So how
do we create this cleanup function? all we need to do is to place a cleanup
function inside useEffect and we just return it that's all we do, we return a
value inside useEffect and that value is a function and then when the component
that uses this `useEffect` or this `useFetch` hook unmounts it fires that
returned cleanup function. So we're gonna do this below setTimeout right here.
*/
  useEffect(() => {
    const abortCont = new AbortController();

    setTimeout(() => {
      //lesson 24 Abort controller argument
      fetch(url, { signal: abortCont.signal })
        .then((res) => {
          // console.log(res);
          if (!res.ok) {
            throw Error("Could not fetch the data for that resource");
          }
          return res.json();
        })
        .then((data) => {
          // console.log(data);
          setData(data);
          // Setting isPending to false once we have the data, therefore
updating the state
          setIsPending(false);
          setError(null);
        })
```

```
        .catch((err) => {
          if (err.name === "AbortError") {
            console.log("fetch aborted");
          } else {
            // set isPending message to false when having an error
            setIsPending(false);
            setError(err.message);
          }
        });
    }, 1000);
```
    /* lesson 24 (2nd read): So we're gonna write return and it's just gonna be
a function and all we do for now is log to the console a message so that we
know when this has run. So we'll go on the browser and refresh and we're going
to go in home page `Home` component, wait for the data and then go to new blog
and we can see the cleanup function run but also if we move quickly away we see
it run again. So it's at this point we actually want to stop the fetch that's
going in the background so that we don't try to update the state. So how do we
do that, to do that we're gonna use something known as the abort controller. So
let's create this abort controller first then we will get the explanation how
to use it. So we'll do that inside useEffect at the top above the setTimeout an
we're gonna store this in a constant and we're gonna call it abortCont (we can
call it whatever we want) and we set it equal to new AbortController (look up
☝🏻) . So we have this abort controller and what we can do with it is we can
associate it with specific fetch request and once we have associated it with a
fetch, we can use the abort controller to stop the fetch and that's what we
wanna do, so the way we associate it with a fetch is by first off adding on a
2nd argument to the fetch which is kind of options for the fetch and then using
the `signal` property like so `fetch(url, {signal})` and we set that equal to
abortCont like `fetch(url, {signal:abortCont.signal})` which is our constant
and that's all there is to it, now we're associating this abort controller with
this fetch and then we can use this now to stop that fetch and we wanna do that
inside this cleanup function below `return () => abortCont.abort()` by
`abortCont.abort()` method like so. So this does is abort whatever fetch is
associated with which is this `fetch(url, { signal: abortCont.signal })` one
and so therefore it's not gonna carry on with the fetch it's going to pause it.
So let's test it out. We still get this error so when we abort a fetch what
happens is the fetch then throws an error an we catch that error in here
`.catch((err) => {
        setIsPending(false);
        setError(err.message);
    }) and when we catch an error we're then updating the state
`setIsPending(false)`
        `setError(err.message)` so we're still updating the state we might
not be updating the data state anymore because we've stopped that fetch but
we're still updating these states (We set isPending, also this error) and
therefore we're still trying to update the whole component with these states so

what we want to do is we want to recognize that error inside this catch block
and if it's a specific type of error if it's an abort error caused by us then
we don't want to update the state, so let's do that if check right here, We're
going to say if err.name === 'AbortError' (see up in catch block), then we
don't want to update the state else if we get different kind of error (like
network error or server error) we wanna update the state (in the else part see
up in the catch block) and still want to let the user know about this error and
we still want to set isPending to false. Iff we abort the fetch in the return
function below and error.name is AbortError then we don't update the state and
we just log this `fetch aborted` to the console so hopefully that will solve
the problem of the component trying to update (running in the background while
being unmounted (solved by the return function) trying to update the state
(solved by if else block inside catch)) and we won't get that error anymore.
And it works! we're removing useEffect,and useState hook import line as we are
not using them in home component */
    return () => abortCont.abort();
  }, [url]);

  return { data, isPending, error };
};
/* Conclusion lesson 24: So that's how we can use the cleanup function and the
abort controller. Next up we're gonna learn Route Parameters🔥 */

export default useFetch;

Lesson 25 (Route Parameters):

Sometimes we need to pass dynamic values as part of a route. In other words a
route where a certain part of it is changeable but regardless of what that
changeable part is it still renders the same page or component. Now an example
of this is for a blog details page so the route might look something like this
`/blogs/123` where 123 is the id of the blog we wanna see now this would maybe
render a `BlogDetails` component and show the blog with the id `123` now this
`123` can easily be something else `456` or `789` now in these cases we'd
render the same `BlogDetails` but instead we'd show the blog with the id of
`456` or `789`. So this changeable part of the route is known as a route
parameter; it's like a variable inside a route. Now in our react application we
need to be able to use route parameters and access those route parameters from
our components so that in the component we can use these ids for example to
then maybe fetch data for that particular blog. So we'll set this up now. Now
we'll create a `BlogDetails` component because this is the component we're
going to show when a user visits a route which looks something like `/blogs/id`
this `id` is the id of the blog.

BlogDetails.js

```
import { useParams } from "react-router-dom";

const BlogDetails = () => {
  /* lesson 25: It allows us to grab route parameters from the route, we need
to destructure whatever route parameter we want, remember we named the route
parameter `id` `:id` so we say we want the `id` (destructure it from object
that useParams returns) from the params then we can access this and we can use
it inside our template or inside some other function to fetch the blog with
that id. So now all we are gonna do to output this in our template */
  const { id } = useParams();
  // lesson 25: we're gonna return a simple template here for now, now we'll
create a route for this blog inside App.js (App component) this is where our
Routes reside.
  return (
    <div className="blog-details">
      {/* lesson 25 Outputting the id grabbed from route parameter. So this id
is visible in our output. so this id is available to us right now and we can
use it to do a fetch request for that blog. But first we want to add links to
blogs in homepage `Home` component so that if a user clicks on them, it goes to
the `BlogDetails` route and it goes to the `id` of whatever this blog is. So
let us go now to the `BlogList` component and set this up  */}
      <h2>Blog Details - {id}</h2>
    </div>
  );
};

export default BlogDetails;
```

App.js

```
import Navbar from "./Navbar";
import Home from "./Home";

import { BrowserRouter as Router, Route, Switch } from "react-router-dom";
// import create component
import Create from "./Create";
import BlogDetails from "./BlogDetails";

function App() {
  // const title = "Welcome to the new blog!";

  return (
    <Router>
      <div className="App">
```

```jsx
      <Navbar />
      <div className="content">
        {/* <Home /> */}
        <Switch>
          <Route exact path="/">
            <Home />
          </Route>
          <Route path="/create">
            <Create />
          </Route>
          {/* lesson 25: this time path is going to `/blogs/'whatever the id
is'`, so how do we say this is gonna be a route parameter a changeable part?
(parameter that is after `/blogs/'Route parameter'`). We can't hard code this
id is it going to be change well to do this we use a colon (:) and we give the
route parameter a name (this is similar what we do in express.js) so we can
call it whatever we want, we're gonna call it id, so that will represent the
`id` of the blog. Now it could be 100 or 124 anything doesn't matter. Now when
a user visits this route we want to we want to go to the `BlogDetails`
component. Now if we test this we should see that route component `BlogDetails`
regardless of what id we put in e.g. `123` in the :id part of the path it works
like a variable (like express.js). example url:
`http://localhost:3000/blogs/123`. This id can be whatever we want it is
changeable renders the `BlogDetails` component. But we need to access whatever
our id `:id` is inside `BlogDetails` component because if it's 1 then we want
to fetch the blog with id of 1 inside this component, if it's 5 we fetch the
blog with id of 5 and so forth. So how do we grab that route parameter right
here in the url `http://localhost:3000/blogs/4` (grab 4 inside BlogDetails
component) well we use a hook from `react-router-dom` */}
          <Route path="/blogs/:id">
            <BlogDetails />
          </Route>
        </Switch>
      </div>
    </div>
  </Router>
);
}


export default App;
```

BlogList.js

```jsx
/* Removing the handleDelete function that was passed as prop from home
component to BlogList component, also the button that is used to invoke it */

import { Link } from "react-router-dom";
```

```
const BlogList = ({ blogs, title }) => {
  return (
    <div className="blog-list">
      {/* lesson 25: Remember inside this function we've access to each
individual blog and each blog has an `id` property so we can access that and we
can create a link. So we're gonna create a `Link` (auto import it) tag in here,
and we're gonna surround `h2` and `p` tag inside the `Link`. Now in the `to`
prop we don't want to hardcode the url cause part of it is going to be dynamic
(the id is changeable for each blog). So we are going to use string literals
(template string) to output the id variable inside curly braces (curly braces
denote we're using some kind of js). Now in this map function we have access to
each `blog` and we'll use the `blog.id` that's the id property in the `blog` in
here. (This similar functionality is used in ejs and express.js as well). So
we're capturing the if in the links and we're accessing these ids inside the
`BlogDetails` component by destructuring the `useParams` hook  */}
      <h2>{title}</h2>
      {blogs.map((blog) => (
        <div className="blog-preview" key={blog.id}>
          <Link to={`/blogs/${blog.id}`}>
            <h2>{blog.title}</h2>
            <p>Written by {blog.author}</p>
          </Link>
          {/* <button onClick={() => handleDelete(blog.id)}>Delete
blog!</button> */}
        </div>
      ))}
    </div>
  );
};
/* lesson 25: We're getting the blogs prop from the `Home` component we're
iterating through the blogs using map function and outputting each blog and
building links for each blog with their specific id that we've access to. */

export default BlogList;
```

----
index.css

```
/* lesson 25: In this css rule we took off the underline style from the anchor
tag, so this blog-preview is the parent of `Link` component/tag that is going
to be an anchor (a) tag in the browser dom  */
.blog-preview a {
  text-decoration: none;
}

/* a:has(h2, p) {
```

```
    text-decoration: none;
} */
-------------------------------------------------------------------
```

Lesson 26 (Reusing a custom hook `with different url argument that is different endpoint`):

Now we can click on one of these links (in `Home` component) to go through to the `BlogDetails` page where we have access to that route parameter, that's the `id` for that blog so we can use that inside this component now to make a fetch request for the details of that particular blog from the `db.json` file and to do that we're gonna be reusing our custom hook `useFetch`, which returns three values the data we're trying to fetch (in our case the individual blog), `isPending` property which is true or false, and also an `error` if there is one. So all we have to do to use this hook in `BlogDetails` component now and pass in the `url` or the endpoint that we want to use to fetch data from. So that's what we're gonna do inside `BlogDetails`.

BlogDetails.js

```
import { useParams } from "react-router-dom";
import useFetch from "./useFetch";

const BlogDetails = () => {
  const { id } = useParams();
  /* lesson 26: destructure data, error, and isPending state from useFetch
hook, we'll use the `id` that we have access to in this component in the
template literal. inside here we'll call this data property blog (data:blog).
So at some point we're gonna have this data, we might have an error, and we
might have true or false for isPending. So we want to conditionally output bit
of template dependant on these values, se we'll get rid of the h2 */
  const {
    data: blog,
    error,
    isPending,
  } = useFetch(`http://localhost:8000/blogs/${id}`);

  return (
    <div className="blog-details">
      {/* lesson 26: <h2>Blog Details - {id}</h2> */}
      {/* lesson 26: Display loading message if isPending is true */}
      {isPending && <div>Content is loading.....</div>}
      {/* lesson 26: Output error if there is one */}
      {error && <div>{error}</div>}
      {/* lesson 26: finally we want a bit of template for the blog itself once
we have, once we have the blog Details, once we have a value for the blog, so
blog that is data in useFetch.js is starts with the value of null, if the fetch
```

in useFetch hook is successful it will return the data (that's truthy value) otherwise upon unsuccessful fetch it will have the starter value of null (falsy value). So this can be displayed using conditional templating. So we're going to return some template in parenthesis */}
```
      {blog && (
        /* lesson 26: So we're outputting all properties of this blog in this
```
article tag. So if we view this in the browser we can directly go to the url or visit the link from homepage (`Home` component) in both cases it works🔥. Now we will add a bit of css to make it little bit better */
```
        <article>
          <h2>{blog.title}</h2>
          <p>Written by {blog.author}</p>
          <div>{blog.body}</div>
        </article>
      )}
    </div>
  );
};
```

/* lesson 26: Conclusion: So we can see now how useful this useFetch hook is cause we're just passing in the endpoint whenever we want to get some data and we're getting these 3 properties back (data (itself), isPending (state that is true or false), error (if there is one) btw all of them are states in useFetch hook so we are returning an object in useFetch hook that makes them properties) every time. So we can reuse this over and over again whatever the endpoint is in different components. */
```
export default BlogDetails;
```
/* Next up we're gonna make forms in create page🔥 */
----
index.css

/* Lesson 26: BlogDetails page */
/* blog details page */
```css
.blog-details h2 {
  font-size: 20px;
  color: #f1356d;
  margin-bottom: 10px;
}
.blog-details div {
  margin: 20px 0;
}
```
/* This button below is going to be used when we add the delete functionality later */
```css
/* .blog-details button {
  background: #f1356d;
  color: #fff;
```

```
  border: 0;
  padding: 8px;
  border-radius: 8px;
  cursor: pointer;
} */
```

Lesson 27 (Controlled Inputs - Forms):

Now we'll turn our attention into web forms in react so a user can type in a
new blog and add that and later we'll send a post request so that so it adds it
to the data, In order to this we need to talk about controlled inputs and
different form fields. So controlled inputs are basically a way in react of
setting up input fields in forms so that we can track their values so if we had
a text input field for example a user can type into it and we can store that
value of what they type in some kind of state and we can also make it so if the
state changes that in turn updates the value that we see in the input field. So
we're always keeping the input field and our state in sync with each other. So
let's try this out so the first thing we're gonna do is create a bit of a
template for this form in our `Create` component.

Create.js

```
import { useState } from "react";

const Create = () => {
  /* lesson 27: (read it last):Required states to keep track of user inputs,
```
the title will have an initial value of empty string '', now we're gonna
associate title state with the value of title input. So we're gonna write in
the title input value={title} see below 👇. Now whatever the value of title is
that is '' (empty string) initially will be the value of title input field.
Like we can test it by giving the initial value 'Hello Folks!' in useState it
will show up in the input field in the browser. Now one thing to notice if we
try to change it in the browser, write anything in the input fields nothing
happens it stays 'Hello Folks' we can't delete it or add anything else cause
it's always showing the initial value of 'Hello Folks', It won't let us change
the value of the input, So wanna make it so when we try to change this it in
the input it triggers the `setTitle` that is the function to update the `title`
state to whatever we wanna change it to, so the way to do this is using the
`onChange` event(See the input after label blog title). We will set this
onChange event equal to an anonymous function, which then invokes `setTitle`.
So we're gonna change the title when we try to change the input value. Now
inside the parameter we get access to the event object as the first parameter
we can call it whatever we want in this case it's `e` `onChange={(e) =>
console.log(e)}`. So we want to update the title state with whatever we type in
the input and we can get that from the event object by writing `e.target` the
target is the title input element itself, and then we'll do `e.target.value` as

we want the the value of this input and update the title state by passing the value (e.target.value) to `setTitle(e.target.value)` function. So e.target.value is whatever we're typing into the input field(e.target) that is the value of it. So if we save this now when we type into the input field it's going to trigger this call `setTitle` with the input value that is going to update the `title` state (that was an empty string to begin with). So it's going to update that everytime we type it in to match what we're typing into the input field and vice-versa the value of input is going to match that as it is equal to `title` so now we have this two way binding. So if we save this and type it works!🔥, but also we want to see that, so we're gonna output this (the `title`) at the bottom in the `p` tag after `button` in `form`. So now we can see the title updating. */
  const [title, setTitle] = useState("");
  /* lesson 27: Creating state for body and author */
  const [body, setBody] = useState("");
  const [author, setAuthor] = useState("mario");

  // This template will be very simple to begin with, Now we will show this component App.js (App Component)
  return (
    <div className="create">
      <h2>Add a new blog</h2>
      {/* lesson27 (start from here): Underneath the h2 we'll add a form, we'll get rid of the action attribute because we're gonna submit the form in another video, the we first of all want a label for our first field */}
      <form>
        {/* lesson 27: get rid of the htmlFor(prop) attribute in label as well */}
        <label>Blog title:</label>
        {/* lesson 27: Under this we'll have an input field of type text, but we're gonna format this a little bit different, we're gonna put all the different property names or attributes on a new line cause there will be quite a lot of them as we go through this and we don't want them go off the page over to the right, so this is just easier for us to see🧑, This input field is for title we'll need another for the content and this will be a textarea and get rid of all the attributes and required, we're gonna add more properties in here later on but for now move on and add final label for author, and for this we're gonna use a select field and this is so we can learn how to use controlled inputs with select as well, so at last add a button as well to submit this form. So the form looks terrible and we'll add few styles to make it look better */}
        <input
          type="text"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
          required

```jsx
      />
      <label>Blog body:</label>
      <textarea
        value={body}
        onChange={(e) => setBody(e.target.value)}
        required
      ></textarea>
      <label>Blog author:</label>
      <select value={author} onChange={(e) => setAuthor(e.target.value)}>
        <option value="mario">mario</option>
        <option value="yoshi">yoshi</option>
        <option value="luigi">luigi</option>
      </select>
      <button>Add blog</button>
      <p>{title}</p>
      <p>{body}</p>
      <p>{author}</p>
    </form>
    {/* lesson 27: So now in the form we can fill in the inputs but currently
when we do that we're not keeping track of what a user types into them or what
a user selects (in the select input). So that's the whole aim of this (keeping
track of what a user types into inputs). If we're to start typing, for example
welcome in the blog title we want to be able to track that value and store it
in some state inside this `Create` component, so we can do something with this
data later on. So how do we do this? First of all we need to setup some states
for this. The states are created at the top above return☝️ */}
    </div>
  );
};
/* lesson 27 (Conclusion): So now we're tracking what a user types into this
field we need do the same though with the other input that is body, So same
thing is done with body as with title. But what about the select field that is
also similar the initial value of the select is going to be the value
predefined in the select options that are mario, yoshi, and luigi. So we'll put
mario as initial value of author state in `useState`. Now on the select field
we'll set the value equal to author state. Now doing similar thing with
onChange event and outputting the author state in the `p` tag. Next up we're
gonna submit this form🔥 */

export default Create;
```

----
index.css

```css
/* lesson 27: create new blog form */
.create {
  max-width: 400px;
```

```
  margin: 0 auto;
  text-align: center;
}
.create label {
  text-align: left;
  display: block;
}
.create h2 {
  font-size: 20px;
  color: #f1356d;
  margin-bottom: 30px;
}
.create input,
.create textarea,
.create select {
  width: 100%;
  padding: 6px 10px;
  margin: 10px 0;
  border: 1px solid #ddd;
  box-sizing: border-box;
  display: block;
}
.create button {
  background: #f1356d;
  color: #fff;
  border: 0;
  padding: 8px;
  border-radius: 8px;
  cursor: pointer;
}
```

Lesson 28 (Submit Events):

Now we're tracking what a user types or selects inside this form next we wanna
handle the form being submitted and eventually do something with this data, now
when a button is pressed inside a form it fires a submit event on the form
itself so we can listen for this submit event and react to it, now another
option would be to attach a click event to the button itself and react to that
instead but we prefer to react to the submit event so what we're gonna to
attach that in the form in `Create` component.

Create.js

```
import { useState } from "react";

const Create = () => {
```

```jsx
  const [title, setTitle] = useState("");
  /* lesson 27: Creating state for body and author */
  const [body, setBody] = useState("");
  const [author, setAuthor] = useState("mario");

  const handleSubmit = (e) => {
    e.preventDefault();
    const blog = { title, body, author };
    console.log(blog);
  };
  return (
    <div className="create">
      <h2>Add a new blog</h2>
      {/* lesson 28: now we'll set this onSubmit equal to something, which is a
function that we'll create up, that is handleSubmit in here we'll take the
event object `e` as parameter that we have access to and we're gonna reference
this function handleSubmit as `onSubmit={handleSubmit}`, so inside this
handleSubmit we want to prevent the default action of the form being submitted
now that default action is for the page to refresh, if we click the `Add blog`
button right now it'll refresh the page, we don't want that to happen, So in
order to prevent that we take this event object(e) and use a method called
`preventDefault` on that and that prevents the page from being refreshing. We
can check in the browser by clicking the `Add blog` button in the browser now
we're going to create the blog object an blog object and blog objet ultimately
going to be the thing that is saved in the db.json file, this object is going
to have title, body, and author property when we're using json-server we don't
need to give the id property cause later when we make a post request
json-server will automatically add an unique id for us, so we don't need to
create that, so we're gonna now create a new object inside `handleSubmit`
called blog and set it equal to an object where we add in the title, body, and
also the author. So this object will have the state variables as properties
that we are taking from the input and updating the state variables themselves.
Then we can log this blog object to the console to see it. So we can now check
the blog object being logged in the browser console. So now we are reacting to
this submit event and we're preventing the default action which is the page
refreshing. Next up we want to send a post request to a particular endpoint
that json server provides us with, so we can add this new object that is `blog`
to our data (db.json file) and we'll do that next */}
      <form onSubmit={handleSubmit}>
        <label>Blog title:</label>
        <input
          type="text"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
          required
        />
```

```
        <label>Blog body:</label>
        <textarea
          value={body}
          onChange={(e) => setBody(e.target.value)}
          required
        ></textarea>
        <label>Blog author:</label>
        <select value={author} onChange={(e) => setAuthor(e.target.value)}>
          <option value="mario">mario</option>
          <option value="yoshi">yoshi</option>
          <option value="luigi">luigi</option>
        </select>
        <button>Add blog</button>
      </form>
    </div>
  );
};

export default Create;
```

Lesson 29 (Making a post request):

Now we can submit the form and create this blog objects, the next step is to
make a post request to json server to add that data to our data file so we're
not gonna use the useFetch hook this time we could probably edit this so we can
handle post requests as well and make a more universal hook but we don't want
do that instead we're just gonna make the post request inside this handleSubmit
function in `Create` component cause we're only gonna make that request once in
the whole application(will not be reusing) and not in different places. First
of all we're going to delete the consol.log(blog) in the `handleSubmit`
function.

Create.js

```
import { useState } from "react";

const Create = () => {
  const [title, setTitle] = useState("");
  const [body, setBody] = useState("");
  const [author, setAuthor] = useState("mario");
  const [isPending, setIsPending] = useState(false);

  const handleSubmit = (e) => {
    e.preventDefault();
    const blog = { title, body, author };
    /* lesson 29 (read second): So let's inside here set isPending to true */
```

```
      setIsPending(true);

      /* lesson 29 (read first): we're gonna make a fetch request using the fetch
api and this is gonna be to the same endpoint as the get request to get all the
blogs so copy the endpoint from the `Home` component where we're
showing/fetching all blogs. Now we will add in 2nd argument in this fetch that
is where we add the data and also define the type of request we're sending here
that is the method, this is gonna be a post request so the method is post, we
also need the headers property and that is so we can the content type is being
sent, so let's say `headers: { "Content-Type": "application/json" }` so this is
basically telling the server the type of content that we're sending with this
request, we're sending json data, now the next property is the body which is
the actual data we're sending with this request and that data is going to be
the `blog` object defined up☝. But we don't just say blog first of all we've
to turn this from an object into a json string to do that we use the `JSON`
object and then we use the method `stringify` and we pass in the object we want
to turn into a json string and that is the `blog` and that's pretty much it. So
this will then make a post request to this endpoint which is the endpoint we
need to make a post request to add in a new blog and that will add a new blog
for us and json server will automatically add the `id` property as well now
since this is asynchronous and it returns a promise we can add a then method
which fires a function when the fetch is complete (the whole post request thing
in fetch) and logs a string in console. So let's test this. So we fill in the
form submit and we'll see the blog being added in the `Home` component. So that
works but we also want to add a loading state when we add the blog and it says
in the button something like loading so let's try that we're going to create
another piece of state up here☝, that is isPending as in useFetch hook and
we're gonna set this to useState and to begin with it's gonna be false cause
when we first load the page we're not making that request straight away, it's
only when we try to submit the form so that's right here in handleSubmit
function were we're making the request   */
      setTimeout(() => {
        fetch("http://localhost:8000/blogs", {
          method: "POST",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify(blog),
        }).then(() => {
          /* lesson 29 (read third): At this point down here we want to set it
equal to false when it's completed, so we can set isPending to false. Okay then
what we like to do is to output one button if isPending is false and that's the
`Add blog` button and a different one if isPending is true cause at that point
we're gonna show something like loading in button or maybe make it disabled so
we can't click it */
          setIsPending(false);
          console.log("new blog added!");
        });
```

```
    }, 1000);
  };
  return (
    <div className="create">
      <h2>Add a new blog</h2>
      <form onSubmit={handleSubmit}>
        <label>Blog title:</label>
        <input
          type="text"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
          required
        />
        <label>Blog body:</label>
        <textarea
          value={body}
          onChange={(e) => setBody(e.target.value)}
          required
        ></textarea>
        <label>Blog author:</label>
        <select value={author} onChange={(e) => setAuthor(e.target.value)}>
          <option value="mario">mario</option>
          <option value="yoshi">yoshi</option>
          <option value="luigi">luigi</option>
        </select>
        {/* lesson 29 (read fourth): So we're gonna surround this button in
curly braces and we're gonna say !isPending. So this is only gonna output when
`isPending` is false. When isPending is false the negation will make it true
the button will show in the browser  */}
        {/* {!isPending && <button>Add blog</button>} */}
        {/* lesson 29 (read fifth): So if isPending is true we're gonna output
the disabled button, and we're gonna say Adding blog... in it */}
        {/* {isPending && <button disabled>Adding blog...</button>} */}
        {/* lesson 29 (read sixth): Alternatively, we can use a ternary
operator. condition ? trueExpression (if condition true) : falseExpression (if
condition false) */}
        {isPending ? (
          <button disabled>Adding blog...</button>
        ) : (
          <button>Add blog</button>
        )}
      </form>
    </div>
  );
};
```

```
/* lesson 29: Conclusion: So now everything works, but it will be nice if we
redirect the user to the homepage `Home` component after the blog has been
added and we'll see how to do these redirect next */
export default Create;
--------------------------------------------------------------------------
```

index.css
```
---------
/* Lesson 29: target the disabled button and make it gray color */

.create form button:disabled {
  background-color: #868e96;
}
--------------------------------------------------------------------------
```

useFetch.js (custom hook) - error fix
```
--------------------------------------
/* lesson 29: fixed an error made by net ninja, he passed true as "true" which
was a truthy string value which make this error undetected fixed in below line
*/
const [isPending, setIsPending] = useState(true);
--------------------------------------------------------------------------
```

Lesson 30 (Programmatic Redirects):

Now we can successfully add new blogs by submitting this form but once we've
added the blog it stays on the new blog page `Create` component and what we'd
like to do instead is to redirect the user back to the home page `Home`
component once it's complete. So in order to that we can use another hook in
react called useHistory, and what this allows us to do is go back and forward
through the history much like the back (<-) and forward (->) arrow buttons in
the browser and also add a new page onto the history in other words redirect
them to a new route so let's see how we can do this, first of all inside the
`Create` component we want to import useHistory.

Create.js
```
---------
import { useState } from "react";
/* lesson 30 (read first): import useHistory from react router package */
import { useHistory } from "react-router-dom";

const Create = () => {
  const [title, setTitle] = useState("");
  const [body, setBody] = useState("");
  const [author, setAuthor] = useState("mario");
  const [isPending, setIsPending] = useState(false);
  /* lesson 30 (read second): Next we need to invoke this `useHistory` hook,
that's all we need to do, now e've this object which represents history and on
that object we can use several methods we can use to either go back through
```

history, go forward through history so that's like clicking the back and forward arrows in the browser, but also redirect the user, so first of all what we'll do is maybe show how to go back through history, so we'll say history and use a method called go (in the `handleSubmit` function). */

```
  const history = useHistory();

  const handleSubmit = (e) => {
    e.preventDefault();
    const blog = { title, body, author };
    setIsPending(true);

    fetch("http://localhost:8000/blogs", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(blog),
    }).then(() => {
      setIsPending(false);
      console.log("new blog added!");
      /* lesson 30 (read third): Now we can pass in a (-) number in here if we
```
want to go backward several steps through like `history.go(-1)` this means we go back one through history and we want to place this inside `then` block right here that executes when fetch post operation is finished. So after we added the blog we then want to go back one, for example, if we go to homepage `Home` and we go to a details page `BlogDetails` the we to new blog page `Create` now if were to add this and submit the form, once the blog is added it should go back one which was to that blog details page `BlogDetails`. So we can cj=heck it in the browser. If we want to go forwards we can change the argument of go method in history to one but in this case that makes no sense, instead we want to do is to redirect the user back to the homepage, now the way we do that by taking the history object and using a push method instead and then all we need to do is place the route url inside, and that is `/` route for the homepage that renders `Home` component. */
```
      history.push("/");
    });
  };
  return (
    <div className="create">
      <h2>Add a new blog</h2>
      <form onSubmit={handleSubmit}>
        <label>Blog title:</label>
        <input
          type="text"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
          required
        />
```

```
        <label>Blog body:</label>
        <textarea
          value={body}
          onChange={(e) => setBody(e.target.value)}
          required
        ></textarea>
        <label>Blog author:</label>
        <select value={author} onChange={(e) => setAuthor(e.target.value)}>
          <option value="mario">mario</option>
          <option value="yoshi">yoshi</option>
          <option value="luigi">luigi</option>
        </select>
        {/* lesson 29 (read fourth): So we're gonna surround this button in
curly braces and we're gonna say !isPending. So this is only gonna output when
`isPending` is false. When isPending is false the negation will make it true
the button will show in the browser  */}
        {/* {!isPending && <button>Add blog</button>} */}
        {/* lesson 29 (read fifth): So if isPending is true we're gonna output
the disabled button, and we're gonna say Adding blog... in it */}
        {/* {isPending && <button disabled>Adding blog...</button>} */}
        {/* lesson 29 (read sixth): Alternatively, we can use a ternary
operator. condition ? trueExpression (if condition true) : falseExpression (if
condition false) */}
        {isPending ? (
          <button disabled>Adding blog...</button>
        ) : (
          <button>Add blog</button>
        )}
      </form>
    </div>
  );
};

/* lesson 30 (Conclusion): So that's all we want to do. The user submits the
form, the blog is added, once it's added in the `then` method we push them to
the homepage. So let's test this. Next up we're going to delete blogs */

export default Create;
```

Lesson 31(Deleting Blogs):

So things are looking good we can successfully list all the blogs, we can add
new blogs, we can also see the blog details as well when we click on the blog
now what we'd like to is to be able to delete a blog by clicking on a button at
the bottom of `BlogDetails` component, So we'll do that now over to

`BlogDetails` now what we're gonna do to place the delete button a the bottom
of the article.

BlogDetails.js

```
import { useHistory, useParams } from "react-router-dom";
import useFetch from "./useFetch";

const BlogDetails = () => {
  const { id } = useParams();
  const {
    data: blog,
    error,
    isPending,
  } = useFetch(`http://localhost:8000/blogs/${id}`);
  const history = useHistory();

  const handleClick = () => {
    /* lesson 30 (read last): So for deleting http://localhost:8000/blogs/${id}
is going to be our endpoint and we have access to the id variable in this
`BlogDetails` function/component that is from the returned object of useParams
hook that we destructured into id, and we have access to `blog.id` as well that
we're getting from the useFetch hook in here, see the useFetch destructuring
above☝️, so that's the endpoint now we need our second argument in the fetch
that is the object where we can say the method this time is going to be DELETE
and that's all we need to do in there, we're not sending any data all we're
doing is asking json server to delete this blog with this `id` `blog.id` in
this case, both are same though. Now we'll chain a then method to this fetch
cause it's asynchronous which is going to fire a function when the fetch is
complete and inside here what we'd like to do is again redirect the user to the
homepage `Home` component once the blog is deleted, cause once the user clicked
the button to delete the blog it makes no sense to stay in the `BlogDetails`
page, we wanna take them back to the homepage, So let's do that by first of all
using the useHistory hook in the `then` block, when the fetch is complete we'll
redirect the user to the homepage using push('/') method to push on a new route
in history object see below. `/` is the homepage route that renders the `Home`
component. So all this should work, check in the browser. So all works, now
we're gonna add some styles for the delete button in our global(the styles
apply to all components) index.css file  */
    fetch(`http://localhost:8000/blogs/${blog.id}`, {
      method: "DELETE",
    }).then(() => {
      history.push("/");
    });
  };
```

```
    return (
      <div className="blog-details">
        {isPending && <div>Content is loading.....</div>}
        {error && <div>{error}</div>}
        {blog && (
          <article>
            <h2>{blog.title}</h2>
            <p>Written by {blog.author}</p>
            <div>{blog.body}</div>
            {/* lesson 31 (read first): We'll place the delete button here, and
we need to attach a click handler(onClick) to this, and we set it equal to
something, now we need to create a function that we can associate with this
button we'll call this `handleClick` and we'll associate that to our onClick
event, So handleClick is going to be fired when we click on this button. So
inside this button in `handleClick` function we wanna make a fetch request
which is gonna be a delete request  */}
            <button onClick={handleClick}>Delete</button>
          </article>
        )}
      </div>
    );
};

/* lesson 31 (Conclusion): That is pretty much it, we've completed the blog
there is one more thing to do that is to set up a 404 page if we go to some
kind of url that doesn't exist in our routes and we'll do that next up */
export default BlogDetails;
```
----
index.css

```
/* lesson 30: Delete button */
.blog-details button {
  background: #f1356d;
  color: #fff;
  border: 0;
  padding: 8px;
  border-radius: 8px;
  cursor: pointer;
}
```

Lesson 32 (404 pages and next steps):

So the final thing to do in this project is to create some kind of 404
component or a not found component that we wanna show when a user tries to go
to some kind of url that doesn't exist cause currently there is no route setup
for that particular url and it isn't showing any content. So let's create a

component for this inside the source folder we will create a file that is Notfound.js.

Notfound.js

```
import { Link } from "react-router-dom";

/* lesson 32: we'll create a stateless functional component sfc.  This will
have a div with class `not-found`. It is going to have an `h2`,`p` tag, and a
Link to send the user back to the homepage. To import the Link component, write
Link by starting with `<` sign. So this is out `Notfound` component all we
wanna do now is set it up so anyone tries to go to the route that doesn't exist
we show that component and the way to do that is by going to App component and
adding in kind of a catch all route(again it's like express.js) for this. So we
need to put this down here at the bottom in the App.js file that is the `App`
component.*/
const Notfound = () => {
  return (
    <div className="not-found">
      <h2>Sorry</h2>
      <p>That page can not be found.</p>
      <Link to="/">Go back to the homepage...</Link>
    </div>
  );
};

export default Notfound;
```

App.js

```
import Navbar from "./Navbar";
import Home from "./Home";

import { BrowserRouter as Router, Route, Switch } from "react-router-dom";
// import create component
import Create from "./Create";
import BlogDetails from "./BlogDetails";
import Notfound from "./Notfound";

function App() {
  // const title = "Welcome to the new blog!";

  return (
    <Router>
      <div className="App">
        <Navbar />
```

```jsx
        <div className="content">
          {/* <Home /> */}
          <Switch>
            <Route exact path="/">
              <Home />
            </Route>
            <Route path="/create">
              <Create />
            </Route>
            <Route path="/blogs/:id">
              <BlogDetails />
            </Route>
            {/* lesson 32: the Notfound component will be rendered here in a
catch all route, so we're gonna add another route. Again start writing
`Notfound` from `<` sign as it will help autoimport it in the top and now we
want the path in this route to be asterisk `*`, that means look catch any other
route, now this goes at the bottom because otherwise it's gonna match that
comes in if it goes at the top. If it goes at the bottom if none of the routes
defined above it match then this `*` one matches regardless it's kind of like a
catch-all route. So if we save it now, upon entering route in the address bar
that doesn't exist in the browser we'll see the `Notfound` component being
rendered. So with this this project is pretty much complete now🔥. */}
            <Route path="*">
              <Notfound />
            </Route>
          </Switch>
        </div>
      </div>
    </Router>
  );
}
/* lesson 32(Conclusion): To further advance in react we can check out the
react docs or advanced react tutorials. learn useRef, useMemo, useReducer, and
useCallback hook. Also edit functionality in this blog can be implemented by
making a fetch request (send the id as prop from `BlogDetails` component to
some `Edit` component and implement the fetch function PUT request there this
solution is an experiment) */

export default App;
```