Another React Lab

Things to understand in this lab:

- Server-side rendering and dynamic loading of components that only a browser can understand.
- Passing props to child components (syntax)
- "Lifting the state" – finding a parent component to connect components and letting child components update the state.
- Making a simple roulette wheel work (using a package for some animation). In this tutorial, "prize number" is the number that comes up on the roulette wheel, while "lucky number" is the number that a user chooses.

Warning: some code snippets in this lab are later re-written. And ChatGPT isn't as good as it is confident (understanding its output makes you smarter than the computer).

Let's play roulette!

We're going to build on some of the code from last week's lab. So if you don't have it, either grab a copy of the solution from Moodle (remember to run *npm install*), or else run

```
npx create-next-app --example with-passport with-passport-app
```

to get some boilerplate code to work with. You could also start with your own code – the roulette wheel will be front-end only to begin with, with the option to add some back-end code, later. That dictates your folder structure. In our example, we'll be using the pages and components folders to build the roulette wheel.

First step is to add a page for the roulette wheel, and make sure it appears on the navbar. The basic code for a roulette wheel could look like this:

```
import Layout from "../components/layout";

const Roulette = () => {
   return (
   <Layout>
     <h1>Play roulette</h1>

     <p>A roulette wheel will live here</p>


   </Layout>
  );
};

export default Roulette;
```

And to add it to the navbar, you need to edit components/header.js. It is up to you whether you want a user to be logged in or not when they access the roulette wheel, but you can either copy, paste and modify the "Home" navbar entry or else copy, paste and modify the "Profile" navbar entry.

We're going to use https://www.npmjs.com/package/react-custom-roulette as our roulette wheel. So install it with npm (remember -s to save it to package.json).
You will need to use dynamic loading (third solution on https://dev.to/vvo/how-to-solve-window-is-not-defined-errors-in-react-and-next-js-5f97 or https://www.youtube.com/watch?v=Hx2UqlhPmnc if you prefer a video description) and declare the wheel as a component because the roulette wheel component should only load client-side, and you'll get a "Window undefined" error otherwise.

You might need to import next/dynamic as in https://www.npmjs.com/package/@next-tools/dynamic.

Copy the wheel component from the npm package documentation, and make it so that the prizeNumber and mustSpin are configurable (https://react.dev/learn/passing-props-to-a-component#).

Code for component/wheel.js could look something like this:

```
import React from 'react'
import { Wheel } from 'react-custom-roulette'

const data = [
  { option: '0', style: { backgroundColor: 'green', textColor: 'black' } },
  { option: '1', style: { backgroundColor: 'white' } },
  { option: '2' },
]


export default ({mustSpin, prizeNumber}) => (
  <>
    <Wheel
      mustStartSpinning={mustSpin}
      prizeNumber={prizeNumber}
      data={data}
    />
  </>
)
```

And incorporating it into roulette.js could look like this:

```
import dynamic from "next/dynamic";

const Wheel = dynamic(
  () => {
    return import("../components/wheel");
  },
  { ssr: false }
);
```

```
<Wheel
        mustSpin={mustSpin}
        prizeNumber={prizeNumber}
      />
```

That is, there is a dynamically loaded Wheel object, declared as a constant and used within the render function. The properties are suggestions and may change later in the lab.

This just gives us a static wheel, but you should be able to load the page and see it. If there are any problems, fix them first. In reality, you'll also need a way to make the wheel spin. Look at the multi-spin example on https://www.npmjs.com/package/react-custom-roulette and see if you can set up the wheel component so that it has a button and spins to pick a winning number. **Note:** it **might** be worth setting the winning number in the parent component (i.e. roulette.js rather than wheel.js) and passing it to the wheel component as a property, but the prizeNumber could be decided on within the wheel component and then use "Lifting the state" (below) to get the prizeNumber from the wheel component to the roulette page.. It **is** worth feeding the "data" component (i.e. the array with all the numbers/options on it) in as a property, too, that way it can be shared across multiple child components.

The idea behind a roulette wheel in a casino is that people pick a lucky number, place a bet on that number and if it comes up, then they "win". We need to add a box for the user to enter their lucky number. Their options are limited by the numbers on the wheel, and those options are limited by the "data" array. A drop-down menu is probably a good UI for limited options (but feel free to do something different). So, if you've declared the wheel component so that the data array is a property, then you can declare the data array in the parent component (the roulette.js file) and pass it to both the wheel and the user input component.

I asked ChatGPT to write a dropdown component in JSX and this is what it produced:

```
import React, { useState } from 'react';

const Dropdown = () => {
  // State to manage the dropdown's open/closed status
  const [isOpen, setIsOpen] = useState(false);

  // State to manage the selected value
  const [selectedValue, setSelectedValue] = useState(null);

  // Dummy options for the dropdown
  const options = ['Option 1', 'Option 2', 'Option 3', 'Option 4'];

  // Function to handle option selection
  const handleSelect = (option) => {
    setSelectedValue(option);
    setIsOpen(false);
  };

  return (
    <div className="dropdown">
      {/* Button to toggle the dropdown */}
      <button onClick={() => setIsOpen(!isOpen)}>
        {selectedValue || 'Select an option'}
      </button>
```

```
      {/* Dropdown menu */}
      {isOpen && (
        <ul>
          {options.map((option, index) => (
            <li key={index} onClick={() => handleSelect(option)}>
              {option}
            </li>
          ))}
        </ul>
      )}
    </div>
  );
};


export default Dropdown;
```

It's not a bad start, but, frankly the code from https://www.simplilearn.com/tutorials/reactjs-tutorial/how-to-create-functional-react-dropdown-menu can be used more easily, and actually works. The things that ChatGPT got right are that you need to have an array of options (which we'll feed from the parent component), and you use a map function to turn those into a dropdown. But for some reason, it has produced the options as just visible list items (
) – they're not actually selectable. Have a go at writing this component (with or without ChatGPT's help) before looking at the possible dropdown.js code below.

You can declare this dropdown as a sub-component, but you will run into a problem: how do you get the user's selected lucky number from the child component (the dropdown) to the page (roulette.js)? You need to compare the user's lucky number (from the dropdown) with the roulette wheel's prize number in order to determine if the user is a winner or not. There are three possible solutions:
1 – Just make it part of the page, don't bother with a sub-component.
2 – "Lift the state": https://react.dev/learn/sharing-state-between-components
3 – Use Redux to share the state across components.
For the sake of "learning the basics properly", we're going to go with option 2. Lifting the state involves feeding a function into the component as a property, and then calling that function within the component. For the dropdown declared in page/roulette.js, this might look something like this:
(additions to state variable of Roulette)

```
const [luckyNumber, setLuckyNumber] = useState(0);
```

(Dropdown displayed within Roulette's render function)

```
    <Dropdown
      data={data}
      selectedValue={luckyNumber}
      setSelectedValue={(e) => setLuckyNumber(e.target.value)}
    />
```

That passes in three properties: the "data" (i.e. the options for the dropdown menu), the previously selected "luckyNumber" (this is now a state variable for Roulette), and

setSelectedValue which, in turn, passes the selected value from the dropdown menu back up into the state of Roulette.
My final code for dropdown.js looks like this:

```
import React from 'react';

const Dropdown = ({data, selectedValue, setSelectedValue}) => {
  // Function to handle option selection
  const handleSelect = (option) => {
    setSelectedValue(option);
  };

  return (
    <div>
      <label>
        Select your lucky number
        <select value={selectedValue} onChange={handleSelect}>
          {data.map((option) => (
            <option value={option.index}>{option.option}</option>
          ))}
        </select>
      </label>
    </div>
  );
};

export default Dropdown;
```

Ultimately, we want to compare the user's lucky number from the dropdown menu with the prize number from the wheel and declare them a winner or not. So it is at this point that you must make the decision about how the prize number should be decided. If you put the code for "randomly" selecting the prize number in the wheel component, then you'll have to lift the state. The code below shows the complete wheel component:

```
import React, { useState } from 'react'
import { Wheel } from 'react-custom-roulette'

export default ({iPrizeNumber, data, setWinningNumber}) => {
  const [mustSpin, setMustSpin] = useState(false);
  const [prizeNumber, setPrizeNumber] = useState(0);

  const handleSpinClick = () => {
    if (!mustSpin) {
      const newPrizeNumber = Math.floor(Math.random() * data.length);
      console.log("wheel prize number: " + newPrizeNumber)
      setPrizeNumber(newPrizeNumber);
      setMustSpin(true);
    }
  }

  return (
```

```
    <>
      <Wheel
        mustStartSpinning={mustSpin}
        prizeNumber={prizeNumber}
        data={data}

        onStopSpinning={() => {
          setMustSpin(false);
          setWinningNumber({value: prizeNumber})
        }}
      />
      <button onClick={handleSpinClick}>SPIN</button>
    </>
  )
}
```

And in turn, when this component is used in pages/roulette.js, it is in the render function and it looks like this:

```
      <Wheel
        iPrizeNumber={prizeNumber}
        data = {data}
        setWinningNumber={(e) => setPrizeNumber(e.value)}
      />
```

A couple of things to note about this code: where is the prize number set? When is the prize number communicated to the parent? Is it before or after the wheel stops spinning?

The last thing to do is to declare some way of communicating with the user whether or not they have "won". This can be done with a function on the main page that looks like this:

```
function IsWinner({ l, p }) {
  console.log("Numbers are: " + l + " " + p)
  if (l==p) {
    return <h1> Winner </h1>;
  } else {
    return <h1> Try again </h1>;
  }
}
```

And it is called from the render function like this:

```
      <IsWinner
         l={luckyNumber}
         p={prizeNumber}
       />
```

Again, a couple of things to note with this code: what happens if the user changes their input after the wheel has spun? Is there a way to fix this fake winning display?

Summary of front-end work and "future work"

In this tutorial so far, you've covered lifting the state so that children components can communicate with parent pages and, indirectly, with each other. Both the lucky number and

the prize number are state variables on the page, but they are changed by the components in response to user input.
The prize number is currently set by one line of Maths in the wheel component. So, every user gets a random prize number. What if we wanted all users to share the same prize number?
At the moment, a "bet" has no "consequences". There's no running total of "cash" won or lost. But it would be relatively straightforward to add a state variable that could indicate the current winnings, and let the user enter an amount to bet.

Adding a back end?

The easiest way for users to have a consistent experience with the roulette wheel (i.e. the winning number is the same for spins that happen at the same time) is to have it controlled by a server. Unfortunately, that means that the server decides when the wheel is spun and calculates the prize number, and the server takes any bets *before* the wheel is spun (maybe storing them in a database of username, bet amount and lucky number). We'll cover some of this in the next lab, but it essentially boils down to writing some API calls and writing some API functionality.

A word about React for production environments

This is really a comment about the scripts available in package.json. You should notice that there are three of them. You've probably been using *npm run dev* to start your server. But really, you should be observing what happens when you run *npm run build* and *npm run start*.

To put React into production, you need to build the production javascript. You might be able to try this with your current build. Try:
npm run build
And take a look at what it produces (clue, it's in the build folder if it is successful). If you see some errors related to profile.js from the previous lab, you can fix these simply by declaring a function that checks to see if "user" is defined before it tries to access user.username and user.profile from within the render function.

Incorporating the build command and running the production level code might be a good idea if you're aiming at a production server. But it is easy enough to know the commands to build and serve the production code: they're in package.json already.

A few more points to note:

- Production and dev servers might link to different databases. If you do this using environment variables, then it's an easy change to make.
- Server-Side-Rendering (or SSR) might cause more problems in production builds than in dev setups. Things that are a warning in dev might be an error in production.
- Some packages won't work in production (sometimes due to SSR, which you can get around with dynamic loading).

Supplemental: The official React tutorial

You should try the React beginner tutorial here: https://beta.reactjs.org/learn/tutorial-tic-tac-toe to gain more understanding of exactly how powerful React is. Do download the **React developer tools** for your browser, too. Since you have completed some React labs in this module, you should already have everything you need for a local React implementation, although you can use the suggested online editors if you wish.