Step by step solution for the Country App:

Exercise Lab: Building a Flag Explorer App in React – Step by Step

Skill Level: Intermediate

Learning Objectives

By completing this lab, students will:

- Understand React's component-based design
- Use the useState and useEffect hooks
- Fetch and display data from an external API
- Render a list of dynamic components
- Implement item removal via state updates
- Calculate and render totals based on state

Technologies

- React (using Create React App or Vite)
- JavaScript (ES6+)
- React Hooks
- Fetch API

Lab Overview
Build a small application that displays information about countries using the REST Countries API, including their name, flag, capital, and population. The App will be able to remove countries from the list of a randomly selected countries and see a live total population count.

Setup Instructions

Step 1: Create Your Basic HTML File

You'll start with the barebones HTML structure.
Create a New File: In an empty folder on your computer, create a new file and name it index.html.
Add Basic HTML Boilerplate: Copy and paste the following code into your **index.html** file.

Step 2: Include React, ReactDOM, and Babel from CDNs

To use React directly in your HTML, you need to include the React library, ReactDOM (which connects React to the DOM), and Babel (to translate modern JavaScript and JSX into browser-understandable code).

1. **Add Script Tags:** Place the following <script> tags **inside the <head> section** of your index.html, just before the closing </head> tag.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Scoreboard</title>
  <link rel="stylesheet" href="css/style.css" />
</head>

<body>
  <div id="root"></div>

  <script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
  <script type="text/babel" src="./js/app.js"></script>
</body>

</html>
```

2. **Clean Up Files**

Remove or clear out App.css and App.js as needed to start fresh.

In the first part of the lab, some data is provided. The data can be defined in the app.js as:

```
countries = [
    {
        "id": "a1b2c3d4",
        "flags": "https://flagcdn.com/w320/tg.png",
        "name": "Togo",
        "capital": "Lomé",
        "population": 8278737
    },
    {
        "id": "e5f6g7h8",
        "flags": "https://flagcdn.com/w320/yt.png",
        "name": "Mayotte",
        "capital": "Mamoudzou",
        "population": 226915
    },
      :
      :
      :
    {
        "id": "g1h2i3j4",
        "flags": "https://flagcdn.com/w320/lu.png",
        "name": "Luxembourg",
        "capital": "Luxembourg",
        "population": 632275
    },
    {
        "id": "k5l6m7n8",
        "flags": "https://flagcdn.com/w320/na.png",
        "name": "Namibia",
        "capital": "Windhoek",
        "population": 2540916
    }
]
```

Step 1: Now write a function to generate an "n" random country array. The following function is based on the Fisher-Yates algorithem:

```
function shuffleArray(array, n = 9) {
    // Create a shallow copy to ensure the original array passed to selectRandomItems
is not mutated
    let newArray = [...array];
    let currentIndex = newArray.length;
    let randomIndex;

    // While there remain elements to shuffle.
    while (currentIndex !== 0) {
        // Pick a remaining element.
        randomIndex = Math.floor(Math.random() * currentIndex);
        currentIndex--;

        // And swap it with the current element.
        [newArray[currentIndex], newArray[randomIndex]] = [
            newArray[randomIndex],
            newArray[currentIndex],
        ];
    }

    newArray = newArray.slice(0, n); // Limit the shuffled array to the first n items
    console.log(newArray);
    return newArray
}


// test the function

shuffleArray(countries, 9);
```

Step 2: Create the core structure of the app and test:

```
const App = () => {
    shuffleArray(countries, 9);

    return (
        <div>
            <h1>Flag Explorer</h1>
            <div class='countries'>
                <div class="country">
                    <h3 class="country-name">Netherlands</h3>
                    <img class="country-flag" src="https://flagcdn.com/nl.svg" />
                    <div class="content">
                        <h3>Capital</h3>
                        <p>Amsterdam</p>
                        <h3>Population</h3>
                        <p>16,655,799</p>
                    </div>
                </div>
            </div>
        </div>
    );
}
```

Step 3: Create a Card component and use it:

```
function Card() {
    return (
        <div class="country">
            <h3 class="country-name">Netherlands</h3>
            <img class="country-flag" src="https://flagcdn.com/nl.svg" />
            <div class="content">
                <h3>Capital</h3>
                <p>Amsterdam</p>
                <h3>Population</h3>
                <p>16,655,799</p>
            </div>
        </div>
    );
}


const App = () => {
    shuffleArray(countries, 9);

    return (
        <div>
            <h1>Flag Explorer</h1>
            <div class='countries'>
                <Card />
            </div>
        </div>
    );
}
```

Step 4: Modify the code to loop through the data:

```
function Card(props) {
    return (
        <div className="country">
            <h3 className="country-name">{props.country}</h3>
            <img className="country-flag" src={props.flag} />
            <div className="content">
                <h3>Capital</h3>
                <p>{props.capital}</p>
                <h3>Population</h3>
                <p>{props.population}</p>
            </div>
        </div>

    );
}


const App = () => {
    let countriesList = shuffleArray(countries, 9);
    // console.log(countriesList);

    return (
        <div>
            <h1>Flag Explorer</h1>
            <div className='countries'>
                {countriesList.map(country => (
                    <Card key={country.id} country={country.name} flag={country.flags}
capital={country.capital} population={country.population} />
                ))}
            </div>
        </div>
    );
}
```

Step 5: Add delete functionality

```
const { useState } = React;

function Card(props) {
    return (
        <div className="country" onDoubleClick={() => props.onRemove(props.id)}>
            <h3 className="country-name">{props.country}</h3>
            <img className="country-flag" src={props.flag} />
            <div className="content">
                <h3>Capital</h3>
                <p>{props.capital}</p>
                <h3>Population</h3>
                <p>{props.population}</p>
            </div>
        </div>

    );
}


const App = () => {

    const [countriesList, setCountriesList] = useState(shuffleArray(countries, 9));

    const removeCountry = (id) => {
        setCountriesList(countriesList.filter(c => c.id !== id));
    };

    return (
        <div>
            <h1>Flag Explorer</h1>
            <div className='countries'>
                {countriesList.map(country => (
                    <Card
                        key={country.id}
                        id={country.id}
                        country={country.name}
                        flag={country.flags}
                        capital={country.capital}
                        population={country.population}
                        onRemove={removeCountry} />
                ))}
            </div>
        </div>
    );
}
```

Test It: Save and refresh your browser.

Now let’s try using fetch API to receive data from public api server. In order to do that, there is another hook called useEffect that we can set to perform a function when a state changes or a page renders.

useEffect

There are different phases in a React component's lifecycle: mounting, updating, and unmounting. React provides lifecycle methods to run code at these specific moments. They are used to re-render a component, update the DOM when data changes, fetch data from an API or perform any necessary cleanup when removing a component. Developers refer to these types of actions as "side effects" because you cannot perform them during rendering (additional code runs after React updates the DOM).

Perform Side Effects in Function Components

React provides the `useEffect` Hook to let you perform side effects in function components, and give them access to common lifecycle hooks.

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

in using CDN you can import useEffect:

```
const { useState , useEffect } = React;
```

In your component, define the `useEffect()` Hook. `useEffect` receives a callback function as the first argument, which is where you perform any side effects:

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

function App() {
 const [score, setScore] = useState(0);
 const [message] = useState('Welcome!');

 // The effect happens after render
 useEffect(() => {
  console.log('useEffect called!');
 });

 return (...);
}
```

The `useEffect` Hook instructs React to do *something* after render, so it's called when the component first renders and after each subsequent re-render or update. If you run the code above, notice how "useEffect called!" immediately displays in the console. The message displays, again and again, each time the `score` state updates.

Prevent `useEffect` from Causing Unnecessary Renders

Since `useEffect` gets called after every render, applying it after each render might create a performance problem. The `useEffect` Hook takes an optional array as a second argument that instructs React to skip applying an effect (and re-rendering) if specific values haven't changed between re-renders.

In the array, you list any dependencies for the effect. In other words, any state variables that are used or updated inside `useEffect`. The array instructs the `useEffect` Hook to run **only** if one of its dependencies changes.

In this example, there are no dependencies; however, passing an empty array as the second argument will run `useEffect` only once after the initial render:

```
function App() {
  const [score, setScore] = useState(0);
  const [message] = useState('Welcome!');

  useEffect(() => {
    console.log('useEffect called!');
  }, []); // pass an empty array to run useEffect once

  return (...);
}
```

Access State Inside `useEffect`

You're able to access a state variable (even update state) from inside `useEffect`. A common example is updating the document's title when state changes. Manually changing the DOM in React components, as shown below, is one example of a side effect:

```
function App() {
  const [score, setScore] = useState(0);
  const [message] = useState('Welcome!');

  useEffect(() => {
    document.title = `${message}. Your score is ${score}`;
  }, []);

  return (...);
}
```

Notice how `useEffect` uses the `score` and `message` variables. This means that the variables are now dependencies that need to be listed inside the array. If you do not list `score`, for example, the title will not display the updated score.

Note: React provides a console warning any time you have missing dependencies.
For example: `React Hook useEffect has missing dependencies: 'message' and 'score'. Either include them or remove the dependency array.`

React runs `useEffect` to compare `score` from the previous render to `score` from the next render. Each time the value of `score` changes (increments/decrements), React re-applies the effect and updates the page's title:

```
function App() {
  const [score, setScore] = useState(0);
  const [message] = useState('Welcome!');

  useEffect(() => {
    document.title = `${message}. Your score is ${score}`;
  }, [message, score]); // add dependencies

  return (...);
}
```

Example: Data Fetching with `useEffect`

You'll most likely use the `useEffect` Hook to fetch data from an API. Consider the following example, which fetches data (a random dog image) from the Dog API:

```
import React, { useState, useEffect } from "react";
function App() {
  const [data, setData] = useState('');

  useEffect(() => {
        console.log('useEffect called!');
        fetch('https://dog.ceo/api/breeds/image/random')
         .then(res => res.json())
         .then(data => setData(data.message))
         .catch(err => console.log('Oh noes!', err))
  }, []);

  return (
   <div className="App">
    <img src={data} alt="A random dog breed" />
   </div>
  );
}

export default App;
```

Again, passing `useEffect` an empty array as the second argument ensures that it runs only once after the component's initial render. In some cases, omitting the second array argument causes `useEffect()` to execute in an infinite loop, endlessly fetching data. This happens because you're modifying the component's state inside `useEffect()`, which triggers the effect again and again.

Resources

- `useEffect` - React docs
- Using the Effect Hook - React docs
- Tip: Use Multiple Effects to Separate Concerns

"Clean Up" `useEffect`

Some side effects in React require "cleanup," or running additional code when a component unmounts (to prevent a memory leak, for example).

With Hooks, you don't need a separate function to perform cleanup. Returning a function from your effect takes care of the cleanup, running the function when the component unmounts. Review the Hooks documentation to learn more about effects with cleanup.

Implementing UseEffect in the Country App:

To continue with our Country App, useEffect can be utilized to setup the countriesList initially:

```
const { useState , useEffect } = React;



const App = () => {

    const [countriesList, setCountriesList] = useState([]);

    useEffect(() => {
        setCountriesList(shuffleArray(countries, 9));
    }, []);

    const removeCountry = (id) => {
        setCountriesList(countriesList.filter(c => c.id !== id));
    };

    return (
        <div>
            <h1>Flag Explorer</h1>
            <div className='countries'>
                {countriesList.map(country => (
                    <Card
                        key={country.id}
                        id={country.id}
                        country={country.name}
                        flag={country.flags}
                        capital={country.capital}
                        population={country.population}
                        onRemove={removeCountry} />
                ))}
            </div>
        </div>
    );
}
```

Fetching Data from an API:

A common use of useEffect is fetching data from public apis. In our case, the country data can be received from https://restcountries.com/v3.1/all?fields=name,capital,population,flags,ccn3

In order to receive the data using useEffect, we should fetch data. The code is in the following:

```
const App = () => {

    const [countriesList, setCountriesList] = useState([]);

    useEffect(() => {
        console.log('useEffect called!');

fetch('https://restcountries.com/v3.1/all?fields=name,capital,population,flags,ccn3')
            .then(res => res.json())
            .then(data => {
                const mappedData = data.map(country => ({
                    id: country.ccn3,
                    name: country.name.common,
                    capital: country.capital ? country.capital[0] : 'N/A',
                    population: country.population,
                    flags: country.flags.svg
                }));
                setCountriesList(shuffleArray(mappedData, 9));
            })
            .catch(err => console.log('Oh noes!', err));
    }, []);

    const removeCountry = (id) => {
        setCountriesList(countriesList.filter(c => c.id !== id));
    };
    return (
        <div>
            <h1>Flag Explorer</h1>
            <div className='countries'>
                {countriesList.map(country => (
                    <Card
                        key={country.id}
                        id={country.id}
                        country={country.name}
                        flag={country.flags}
                        capital={country.capital}
                        population={country.population}
                        onRemove={removeCountry} />
                ))}
            </div>
        </div>
    );
}
```

Now, let’s add total population calculation to when the frame is rendered. For that we add a new UseEffect that is executed everytime countriesList is changed:

```
const App = () => {

    const [countriesList, setCountriesList] = useState([]);
    const [totalpopulation, setTotalPopulation] = useState(0);

    useEffect(() => {
        console.log('useEffect called!');

fetch('https://restcountries.com/v3.1/all?fields=name,capital,population,flags,ccn3')
            .then(res => res.json())
            .then(data => {
                const mappedData = data.map(country => ({
                    id: country.ccn3,
                    name: country.name.common,
                    capital: country.capital ? country.capital[0] : 'N/A',
                    population: country.population,
                    flags: country.flags.svg
                }));

                setCountriesList(shuffleArray(mappedData, 9));

            })
            .catch(err => console.log('Oh noes!', err));
    }, []);

    useEffect(() => {
        console.log('useEffect for total population called!');

        let total = 0; // Initialize a variable to hold the sum

        for (const country of countriesList) {
            total += country.population; // Add each country's population to the total
        }

        // const total = countriesList.reduce((acc, country) => acc +
country.population, 0);
        setTotalPopulation(total);
        console.log('Total population:', total);
    }, [countriesList]);

    const removeCountry = (id) => {
        setCountriesList(countriesList.filter(c => c.id !== id));
    };

    return (
        <div>
            <h1>Flag Explorer</h1>
            <div className='countries'>
                {countriesList.map(country => (
                    <Card
                        key={country.id}
                        id={country.id}
                        country={country.name}
                        flag={country.flags}
                        capital={country.capital}
                        population={country.population}
                        onRemove={removeCountry} />
                ))}
            </div>
            <div className='countries'>
                <h2>Total Population: {totalpopulation.toLocaleString()}</h2>
            </div>
        </div>
    );
}
```

The App now should be in good shape.
Test it in your browser and try to debug if you see any mistakes or anomalies.