

CS 628: Full-Stack Development – Web App

City University of Seattle
School of Technology & Computing
Professor Sam Chung

HOS04: React – Interaction & State

Samantha Hipple
July 25, 2023

PLEASE NOTE

Screenshots in this guide may differ from your environment (e.g., directory paths, version numbers, etc.). When choosing between a stable or most recent release, we advise you install the stable release rather than the best-testing version. Additionally, there may be subtle discrepancies along the steps, please use your best judgment to complete the tutorial. If you are unfamiliar with terminal, command line, and bash scripts, we recommend watching [this video](#) prior to moving forward with this guide. Not all steps are fully explained. Lastly, we advise that you avoid copy-pasting code directly from the guide or GitHub repositories. Instead, type out the code yourself to improve familiarity.

More information on this guide can be found under the related module in [this repository](#).

SECTION CONTENTS

1. Accessing GitHub Codespaces
 2. Event handling in React
 3. Understanding state in React components
 4. Snapshot of state
 5. Queuing and managing state updates
 6. Updating objects in state
 7. Updating arrays in state
 8. Reacting to user input with state
 9. Choosing the right state structure
 10. Sharing state between components
-

SECTION 1. ACCESSING GITHUB CODESPACES

GitHub Codespaces is an online cloud-based development environment that allows users to easily write, run and debug code. Codespaces is fully integrated with your GitHub repository and provides a seamless experience for developers. In order to access Codespaces, users only need a GitHub account and an active internet connection.

After downloading the current HOS assignment, in the top-right corner of the repo, click on the **<> Code** drop-down menu and select **Create codespace on main** as shown in the following image. The free and pro GitHub subscriptions include free use of GitHub Codespaces *up to a fixed*

amount of usage each month. In order to avoid unexpected charges, please review the [billing information](#).

SECTION 2. EVENT HANDLING IN REACT

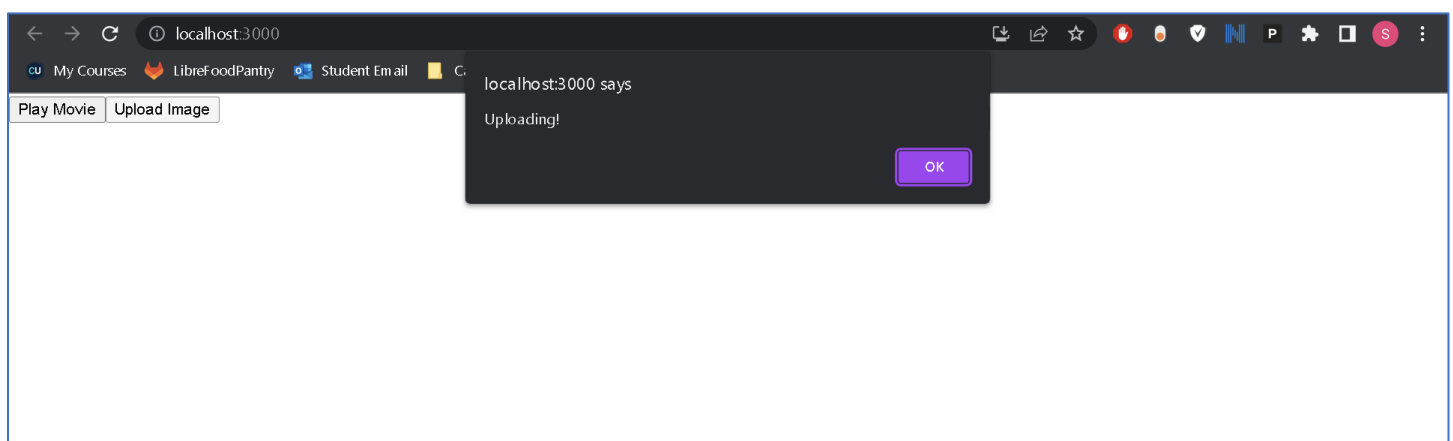
In React, event handlers are essential for creating interactive applications, allowing you to respond to user interactions like clicks, hovers, and form input focus. By attaching custom functions as event handlers, you can define specific behavior for your components.

In the following React example, we have an **App** component that renders a **Toolbar** component, containing two **Button** components. These **Button** components receive event handler props, such as **onPlayMovie** and **onUploadImage**. When a user clicks on either button, an alert is triggered, highlighting how event handlers can be seamlessly passed as props between components in React, enabling personalized responses to user actions.

1. Use **create-react-app** to get a template React webapp up and running.
2. Update **App.js** to match the following:

```
const Button = ({ onClick, children }) => (  
  <button onClick={onClick}>{children}</button>  
);  
  
const Toolbar = ({ onPlayMovie, onUploadImage }) => (  
  <div>  
    <Button onClick={onPlayMovie}>Play Movie</Button>  
    <Button onClick={onUploadImage}>Upload Image</Button>  
  </div>  
);  
  
const App = () => (  
  <Toolbar  
    onPlayMovie={() => alert('Playing!')}  
    onUploadImage={() => alert('Uploading!')} />  
);  
  
export default App;
```

3. Test the app in the browser and save a screenshot of the results.



SECTION 3. UNDERSTANDING STATE IN REACT COMPONENTS

In React, components use state to retain and modify information based on user interactions. The **useState** Hook is a powerful tool that allows you to declare and manage state variables within components, facilitating seamless updates and tracking of the current state values.

Example: Below is a code snippet that demonstrates a React component named **Gallery**, which effectively utilizes the **useState** Hook to manage two state variables: **index** and **showMore**. The **index** variable keeps track of the currently displayed sculpture, while **showMore** determines whether additional details about the sculpture should be visible or hidden. The component also incorporates event handlers (**handleNextClick** and **handleMoreClick**) that update the state in response to user interactions.

1. Create a new file called **data.js** under **src** and add [the code found here](#).
2. Now, update **App.js** to match the following:

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

import { sculptureList } from './data';

const App = () => {
  const [ index, setIndex ] = useState(0);
  const [ showMore, setShowMore ] = useState(false);

  const hasNext = index < sculptureList.length - 1;

  const handleNextClick = () => { hasNext ? setIndex(index + 1) : setIndex(0) }
  const handleMoreClick = () => { setShowMore(!showMore) }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleNextClick}>Next</button>
      <h2>
        <i>{sculpture.name}</i> by {sculpture.artist}
      </h2>
      <h3>{index + 1} of {sculptureList.length}</h3>
      <button onClick={handleMoreClick}>{showMore ? 'Hide' : 'Show'} details</button>
      {showMore && <p>{sculpture.description}</p>}
      <img src={sculpture.url} alt={sculpture.alt}/>
    </>
  );
}

export default App;
```

3. Test the app in the browser and save a screenshot of the results.

localhost:3000

My Courses LibreFoodPantry Student Email Capstone References


Next

Eternal Presence by John Woodrow Wilson

(3 of 12)

Hide details

Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."



SECTION 4. SNAPSHOT OF STATE

State, as a snapshot, refers to the act of capturing and preserving the current data within a React component, making it available for future reference and usage.

Below is an example that illustrates the concept of state as a snapshot in React. The **Form** component effectively manages two state variables: **to** and **message**. The **to** state represents the recipient of the message, while the **message** state stores the content of the message. Whenever the user selects a recipient or types a message, the state is promptly updated through the respective **setTo** and **setMessage** functions. When the user submits the form, the current snapshot of the state is captured and utilized to display an alert message after a delay of 5 seconds, confirming the user's chosen recipient and message.

1. Update **App.js** to match the following:

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

const App = () => {
  const [ to, setTo ] = useState('Alice');
  const [ message, setMessage ] = useState('Hello');

  const handleSubmit = (event) => {
    event.preventDefault();
    setTimeout(() => { alert(`You said ${message} to ${to}`); }, 5000);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        To: { ' ' }
        <select value={to} onChange={e => setTo(e.target.value)}>
```

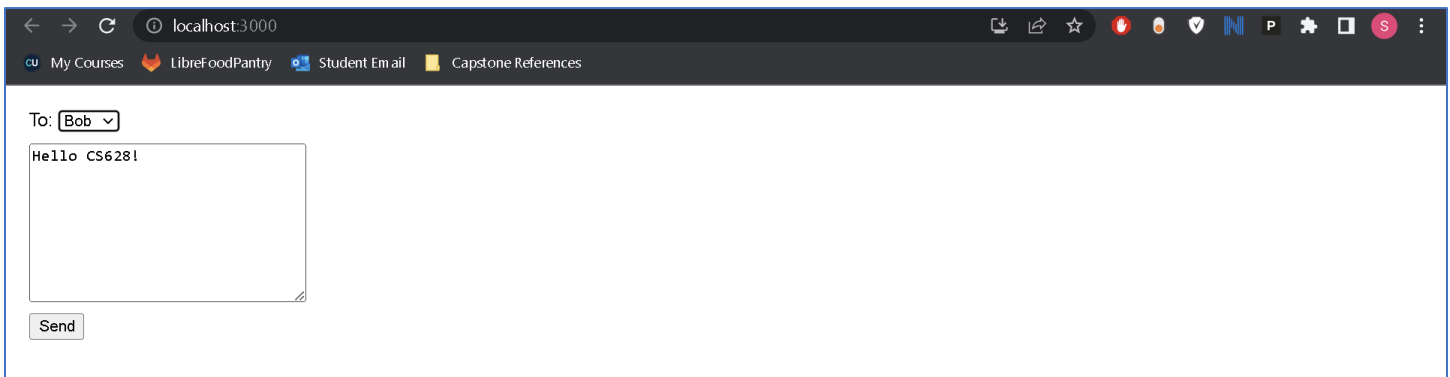
```

      <option value='Alice'>Alice</option>
      <option value='Bob'>Bob</option>
    </select>
  </label>
  <textarea
    placeholder='Message'
    value={message}
    onChange={e => setMessage(e.target.value)} />
  <button type='submit'>Send</button>
</form>
);
}

export default App;

```

2. Replace the styling in **index.css** with [the code found here](#).
3. Test the app in the browser by selecting Bob and entering a custom message.
4. Save a screenshot of the results.



SECTION 5. QUEUING AND MANAGING STATE UPDATES

1. Update **App.js** to match the following:

```

import { useState } from 'react';

const App = () => {
  const [ score, setScore ] = useState(0);

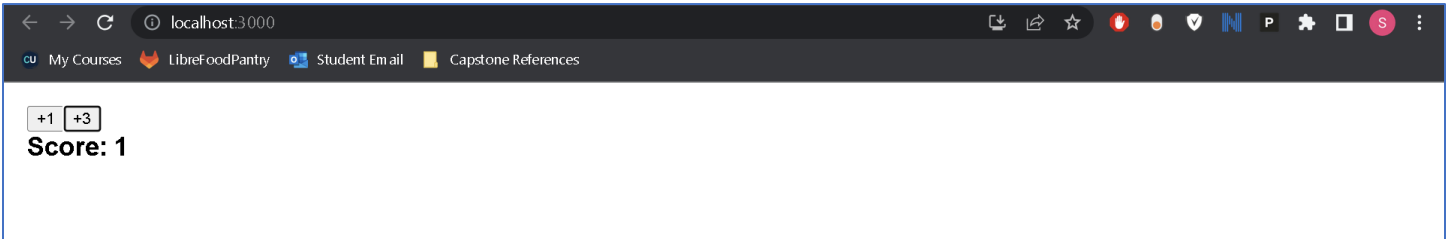
  const increment = () => { setScore(score + 1) }

  return (
    <>
      <button onClick={() => increment()}>>+1</button>
      <button onClick={() => {
        increment();
        increment();
        increment();
      }}>>+3</button>
      <h1>Score: {score}</h1>
    </>
  )
}

```

```
    );  
  }  
  
  export default App;
```

2. Test the app in the browser by clicking the **+3** button once.

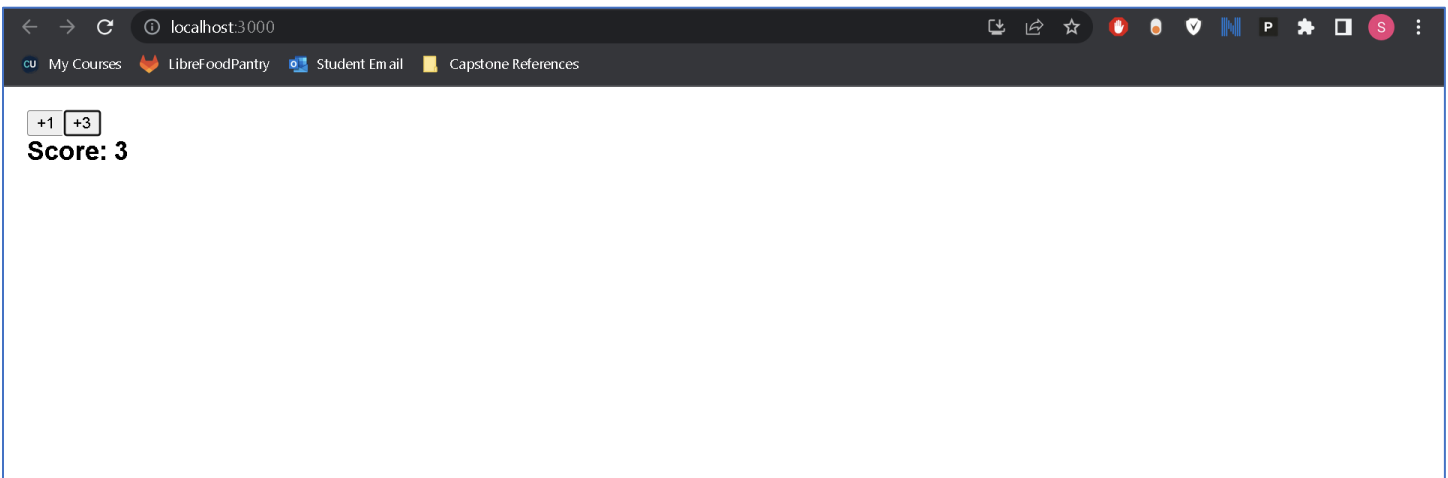


The count should have (*incorrectly*) increased by one instead of three. When setting state in React, it triggers a re-rendering of the component but does not immediately change the value in the existing running code. Therefore, if you call **setScore(score + 1)**, the score will still remain as 0 in the subsequent code right after the call. This issue is easily addressed with the use of an updater function when setting the state.

3. Update the **increment()** function with the following code to fix the **+3** button:

```
const increment = () => { setScore(s => s + 1) }
```

4. Test the app in the browser and save a screenshot.



SECTION 6. UPDATING OBJECTS IN STATE

When working with state in React, it is important to avoid directly modifying objects and arrays stored within the state. Instead, it is recommended to create new objects or make copies of existing ones when updating the state. The spread syntax (...) is commonly used to create copies of objects and arrays that need to be modified. By following this approach, you can ensure that state updates are properly handled and do not result in unintended side effects.

1. Update **App.js** with the following:

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

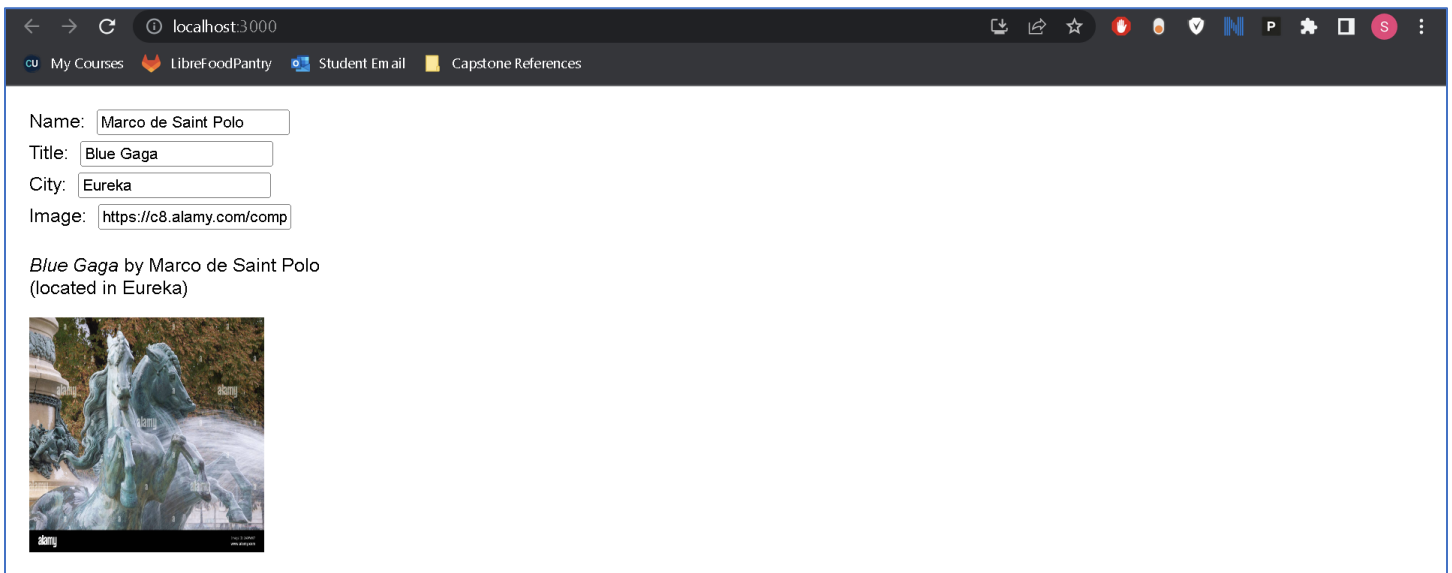
const App = () => {
  const [ person, setPerson ] = useState({
    name: 'Niki de Saint Phalle',
    artwork: {
      title: 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgUOm.jpg'
    }
  });

  const handleNameChange = (e) => { setPerson({ ...person, name: e.target.value }); }
  const handleDetailChanges = (label, e) => { setPerson({ ...person, artwork: { ...person.artwork, [label]: e.target.value } }); }

  return (
    <>
      <label>Name: <input value={person.name} onChange={handleNameChange} /></label>
      <label>Title: <input value={person.artwork.title} onChange={e => handleDetailChanges('title', e)} /></label>
      <label>City: <input value={person.artwork.city} onChange={e => handleDetailChanges('city', e)} /></label>
      <label>Image: <input value={person.artwork.image} onChange={e => handleDetailChanges('image', e)} /></label>
      <p><i>{person.artwork.title}</i> by {person.name}<br />(located in {person.artwork.city})</p>
      <img src={person.artwork.image} alt={person.artwork.title} />
    </>
  );
}

export default App;
```

2. Replace the styling in **index.css** with [the code found here](#).
3. Test the app in the browser by updating the fields.
4. Save a screenshot of the results.



SECTION 7. UPDATING ARRAYS IN STATE

In React, arrays are mutable JavaScript objects that can be stored in state, but it is recommended to treat them as read-only. Like objects, when updating an array stored in state, it is necessary to create a new array or make a copy of the existing one, and then update the state to use the new array.

1. Update **App.js** to match the following:

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

const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: false }
]

const ItemList = ({ artworks, onToggle }) => (
  <ul>
    {artworks.map(artwork =>
      <li key={artwork.id}>
        <label>
          <input
            type='checkbox'
            checked={artwork.seen}
            onChange={e => {onToggle(artwork.id, e.target.checked)}} />
          {artwork.title}
        </label>
      </li>
    )}
  </ul>
);

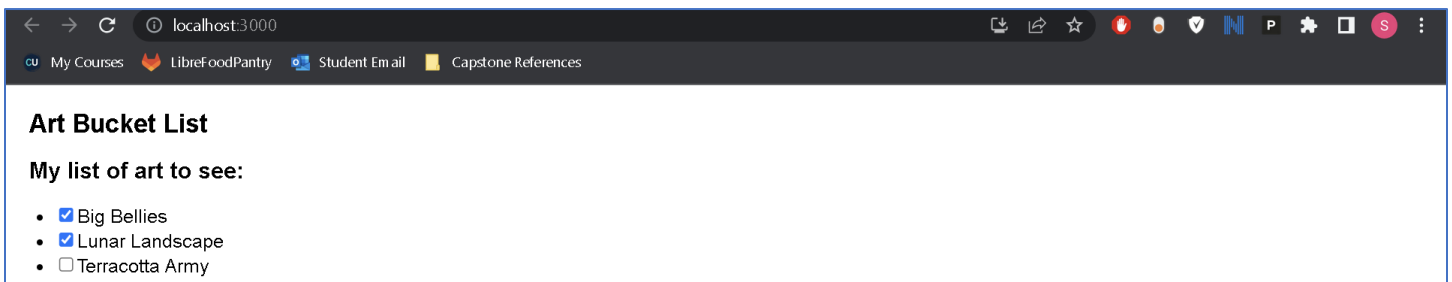
const App = () => {
  const [ list, setList ] = useState(initialList);

  const handleToggle = (artworkId, nextSeen) => {
    setList(list.map(artwork => (artwork.id === artworkId) ? { ...artwork, seen: nextSeen } : artwork));
  }

  return (
    <>
      <h1>Art Bucket List</h1>
      <h2>My list of art to see:</h2>
      <ItemList
        artworks={list}
        onToggle={handleToggle} />
    </>
  );
}

export default App;
```

2. Test the app in the browser by selecting different list items.
3. Save a screenshot of the results.



SECTION 8. REACTING TO USER INPUT WITH STATE

In React, you focus on describing the desired UI for different component states and triggering state changes in response to user input, rather than directly manipulating the UI through code, resembling the approach designers take when thinking about UI.

Observe the implementation of a React quiz form, where the use of the **status** state variable is to determine the enabling or disabling of the submit button, as well as displaying the success message accordingly.

1. Update **App.js** to match the following:

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

const submitForm = (answer) => {
  new Promise((resolve, reject) => {
    setTimeout(() => {
      let shouldError = answer.toLowerCase() !== 'lima';
      if (shouldError) {
        reject(new Error('Good guess, but a wrong answer. Try again!'));
      } else {
        resolve();
      }
    }, 1500);
  })
};

const App = () => {
  const [ answer, setAnswer ] = useState('');
  const [ error, setError ] = useState(null);
  const [ status, setStatus ] = useState('typing');

  const handleSubmit = async (e) => {
    e.preventDefault();
    setStatus('submitting');
    try { await submitForm(answer); setStatus('success') }
    catch (error) { setStatus('typing'); setError(error); }
  }

  const handleTextareaChange = (e) => { setAnswer(e.target.value); }

  return (
    <>
      { status === 'success' ? (
        <h1>That's right!</h1>
      ) : (
        <>
          <h2>City Quiz</h2>
          <p>In which city is there a billboard that turns air into drinkable water?</p>
          <form onSubmit={handleSubmit}>
            <textarea
              value={answer}
              onChange={handleTextareaChange}
              disabled={status==='submitting'} />
            <br />
            <button disabled={answer.length===0 || status==='submitting'}>Submit</button>
          </form>
        </>
      ) }
    </>
  )
}
```

```

      {error != null && <p className='Error'>{error.message}</p>}
    </form>
  </> )}
</>
);
}

export default App;

```

2. Replace the styling in **index.css** with [the code found here](#).
3. Test the app in the browser by submitting a city name.
4. Save a screenshot of the results.

City Quiz

In which city is there a billboard that turns air into drinkable water?

bingo

Submit

Good guess, but a wrong answer. Try again!

SECTION 9. CHOOSE THE RIGHT STATE STRUCTURE

Properly structuring state is crucial for a well-maintained and bug-free component, as it ensures the elimination of redundant information, preventing update-related bugs, facilitating easy modification, and debugging.

1. Update **App.js** to match the following:

```

import { useState } from 'react';

const App = () => {
  const [ firstName, setFirstName ] = useState('');
  const [ lastName, setLastName ] = useState('');
  const [ fullName, setFullName ] = useState('');

  const handleFirstNameChange = (e) => {
    setFirstName(e.target.value);
    setFullName(e.target.value + ' ' + lastName)
  }

  const handleLastNameChange = (e) => {
    setLastName(e.target.value);
    setFullName(firstName + ' ' + e.target.value);
  }

  return (
    <>
    <h2>Let's check you in!</h2>

```

```

    <label>First name:{' '}<input value={firstName} onChange={handleFirstNameChange} /></label>
    <label>Last name:{' '}<input value={lastName} onChange={handleLastNameChange} /></label>
    <p>Your ticket will be issued to: <b>{fullName}</b></p>
  </>
);
}

export default App;

```

2. Test the app in the browser by entering a first and last name.

We can simplify this code by removing the **fullName** variable from state.

3. Replace the code in **App.js** with the following:

```

import { useState } from 'react';

const App = () => {
  const [ firstName, setFirstName ] = useState('');
  const [ lastName, setLastName ] = useState('');

  const fullName = firstName + ' ' + lastName;

  const handleFirstNameChange = (e) => { setFirstName(e.target.value); }
  const handleLastNameChange = (e) => { setLastName(e.target.value); }

  return (
    <>
      <h2>Let's check you in!</h2>
      <label>First name:{' '}<input value={firstName} onChange={handleFirstNameChange} /></label>
      <label>Last name:{' '}<input value={lastName} onChange={handleLastNameChange} /></label>
      <p>Your ticket will be issued to: <b>{fullName}</b></p>
    </>
  );
}

export default App;

```

4. Test the app in the browser by submitting a first and last name.

Notice that the app still functions correctly.

5. Save a screenshot of the results.

Let's check you in!

First name: Last name:

Your ticket will be issued to: **Darius Morgan <3**

SECTION 10. SHARING STATE BETWEEN COMPONENTS

In certain cases, when you need two components to consistently update their state together, you can achieve this by removing the state from both components, placing it in their closest shared parent component, and passing it down as props. This approach, known as **lifting state up**, is a frequently used practice in React development. Rather than maintaining the active state within each individual panel, the state is managed by the parent component, which then provides the necessary props to its children components.

1. Update **App.js** to match the following:

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

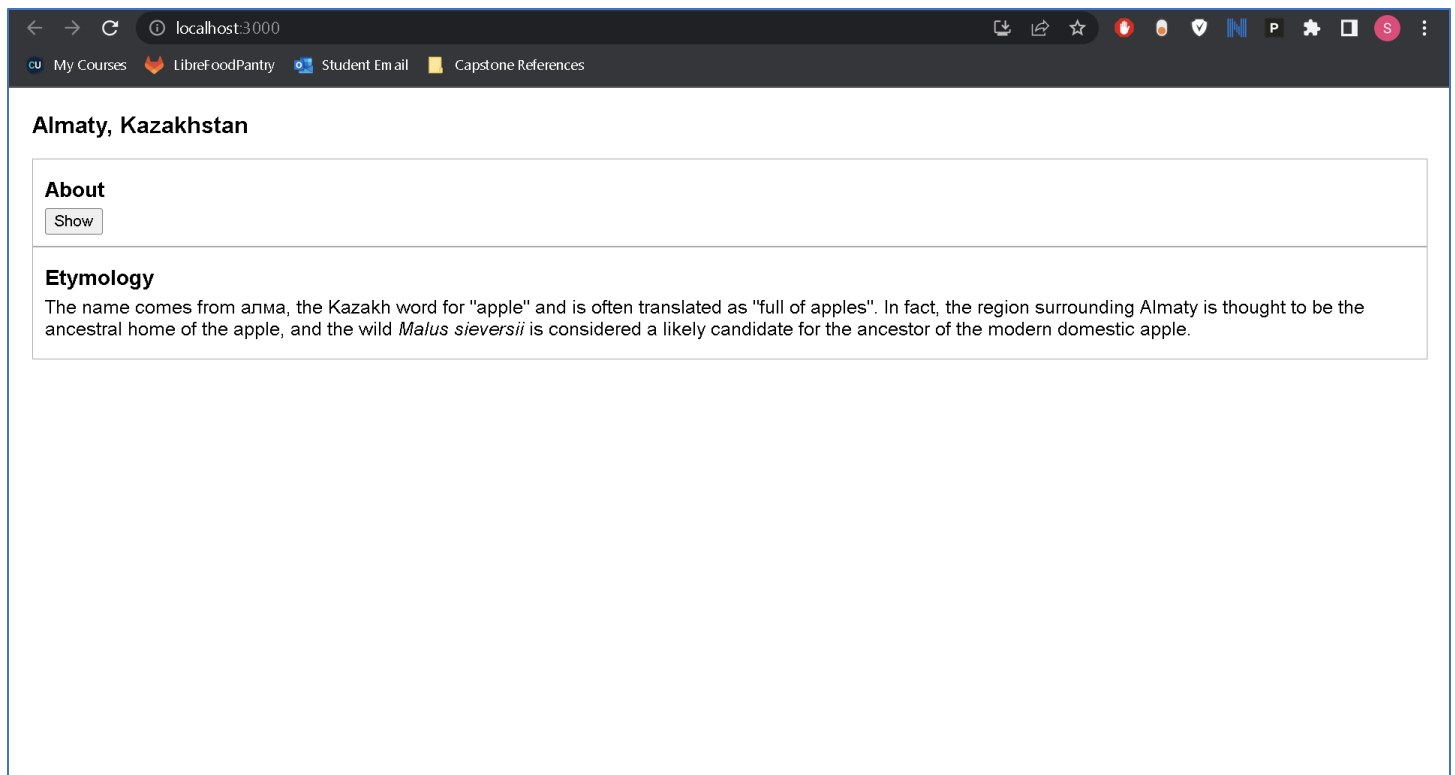
const Panel = ({ title, children, isActive, onShow }) => (
  <section className='panel'>
    <h3>{title}</h3>
    { isActive ? ( <p>{children}</p> ) : ( <button onClick={onShow}>Show</button> ) }
  </section>
)

const App = () => {
  const [ activeIndex, setActiveIndex ] = useState(0);

  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel
        title="About"
        isActive={activeIndex===0}
        onShow={() => setActiveIndex(0)}>
        With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997,
        Almaty was the capity city of Kazakhstan.
      </Panel>
      <Panel
        title="Etymology"
        isActive={activeIndex === 1}
        onShow={() => setActiveIndex(1)}>
        The name comes from <span lang="kk-KZ">алма</span>, the Kazakh word for "apple" and is often
        translated as "full of apples". In fact, the region surrounding Almaty is thought to be the
        ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely
        candidate for the ancestor of the modern domestic apple.
      </Panel>
    </>
  );
}

export default App;
```

2. Replace the styling in **index.css** with [the code found here](#).
3. Test the app in the browser by clicking on the show buttons.
4. Save a screenshot of the results.



PUSH YOUR WORK TO GITHUB TO SUBMIT