Week 4

Welcome!

Goals

- [ ] Discuss TypeScript essentials and how we should structure a component
- [ ] Build the Kudos form with `KudosForm.tsx`

Quick Recap

Last week we learned:

- CSS fundamentals
- How to build the KudosCard component
- How to display multiple cards using sample data

A Quick Note...

The nest few sessions (weeks) will be dedicated to implementing `KudosForm` and the respective cards that are created after a submission is made.

This will take some time, so I decided to split it up into a few parts (3, to be exact). Part 1 covers the TypeScript aspect (this doc), part 2 dives into CSS by having you complete a challenge (week 5), and part 3 is about implementing state, also through a challenge (week 6). Before moving ahead and reading this doc, make sure you've read week 3 as you'll need it for the challenges that lie ahead... □□

Good luck!

Building the Kudos Form Component

Step 1: Setting Up the KudosForm File

First, let's create a new file for our form component.

We will start with just the imports and the component shell:

CREATE: `src/components/KudosForm.tsx`

```
import { useState } from 'react';
import './KudosForm.css';

export function KudosForm() {
  return (
    <div>
      <h2>Form goes here</h2>
    </div>
  );
}
```

What's happening:

- We import `useState` from React - this is the **hook** we'll use for state
- We import our CSS file (which we'll create next)
- We export a function component called `KudosForm`
- For now, it just returns a simple div with text

Step 2: Defining the Props Interface

In this step, we are going to learn about a few key React topics so be sure to ask questions if you are confused.

Currently, we know that our form needs some way to communicate with its parent component (App.tsx).

So, how can we do this? How can we connect the live state of our component - in this case, our kudos form - with its parent component?

To facilitate this communication, we use **props**.

Props

Props are **read-only properties** that are passed from a **parent component** to a **child component**.

They are a fundamental concept for passing data and configuration down the component tree and ensuring that components can talk to each other in real-time.

Interfaces

This may seem like a lot to manage - and it is - but thankfully, React makes it really easy for us to organize and manage this data. To show this, let's talk a bit about interfaces.

Interfaces are primarily used to **define the structure and types** that the props of a component's object must adhere to.

It is a little tricky to understand at first, but it is important to note that interfaces do not actually recieve or store this data, but rather, they enforce 'rules' that an object must follow.

An interface is basically a blueprint (hahahaha□□□□) that describes the data we want to receive from objects.

Destructuring

To make processing this data easier, we can employ a technique called destructuring.

Destructuring allows us to unpack an object's props into easy-to-use variables that we can call right in the function.

Without destructuring, our code looks like this:

```
//Regular Function component
function Welcome(props) {
  return <h1>Hello, {props.name} from {props.city}!</h1>;
}
```

See how we have to use dot-notation to retrieve information from an object? We are making it harder for ourselves than it has to be.

By destructuring objects, we can simply take their distinct variables as parameters rather than calling them from an object.

It looks like this:

```
// Destructuring { name, city } from the props object
function Welcome({ name, city }) {
  return <h1>Hello, {name} from {city}!</h1>;
}
```

Putting these ideas together...

Let's define what data it expects and what it will send eventually send back.

UPDATE: Add this interface at the top of `src/components/KudosForm.tsx`:

```
import { useState } from 'react';
import './KudosForm.css';

// Define what props this component accepts
interface KudosFormProps {
  onSubmit: (kudos: {
    recipient: string; // recipient name as a string
    message: string; // kudos message as a string
    giver: string; // kudos giver as a string
    type: 'kudos' | 'feedback'; // type (kudos or feedback)
    date: string; // date as a string
  }) => void;
}

export function KudosForm({ onSubmit }: KudosFormProps) {
  return (
    <div>
      <h2>Form goes here</h2>
    </div>
  );
}
```

What's happening:

- `KudosFormProps` is a TypeScript **interface** that defines what props our component expects
- `onSubmit` - when the user submits the form, we'll create a kudos object and pass it to the parent component using this function. The parent decides what to do with it (like adding it to a list).
- The function accepts a kudos object with all the fields we need to track
- `type: 'kudos' | 'feedback'` means type can ONLY be one of these two strings

- `void` means the function doesn't return anything
- `{ onSubmit }: KudosFormProps` - we **destructure** the props to get the onSubmit function

---

Step 3: Creating the Form Structure - Header

Let's build the actual form, starting with the header and structure.

UPDATE: Replace the return statement in `src/components/KudosForm.tsx`:

```
export function KudosForm({ onSubmit }: KudosFormProps) {
  return (
    <form className="kudos-form">
      <h2 className="form-title">□ Give Kudos</h2>
    </form>
  );
}
```

What's happening:

- We use `<form>` instead of `<div>` - this is the HTML element for forms
- `className="kudos-form"` - this is how we attach CSS classes in React (remember: not `class`, but `className`)
- The emoji □ gives it some swag
- `form-title` is a CSS class we'll style later

Step 4: Adding the Recipient Input Field

Finally, let's add our first input field!

This will be for the main input we are handling in this program... a Kudos message!

And again, feel free to ask questions as we walk through this - it is super important.

UPDATE: Add the first form group inside the `<form>` in `src/components/KudosForm.tsx`:

```
export function KudosForm({ onSubmit }: KudosFormProps) {
  return (
    <form className="kudos-form">
```

```
      <h2 className="form-title">🎉 Give Kudos</h2>

      <div className="form-group">
        <label htmlFor="recipient">To:</label>
        <input
          id="recipient"
          type="text"
          placeholder="Enter recipient name"
          required
        />
      </div>
    </form>
  );
}
```

What's happening:

- `<div className="form-group">` - A container for the label and input
- `<label htmlFor="recipient">` - The label describes what the input is for
  - `htmlFor="recipient"` connects this label to the input with `id="recipient"`
  - When you click the label, it focuses the input!
- `<input>` - The actual text field
  - `id="recipient"` - Unique identifier, connects to the label
  - `type="text"` - Makes it a text input
  - `placeholder` - The gray text that shows when empty
  - `required` - HTML validation - form won't submit if empty

Step 5: Adding the Message Text area

Next, let's add a textarea for the kudos message.

UPDATE: Add the message field in `src/components/KudosForm.tsx`:

```
export function KudosForm({ onSubmit }: KudosFormProps) {
  return (
    <form className="kudos-form">
      <h2 className="form-title">🎉 Give Kudos</h2>

      <div className="form-group">
        <label htmlFor="recipient">To:</label>
```

```
        <input
          id="recipient"
          type="text"
          placeholder="Enter recipient name"
          required
        />
      </div>

      <div className="form-group">
        <label htmlFor="message">Message:</label>
        <textarea
          id="message"
          placeholder="Write your kudos or feedback..."
          rows={4}
          required
        />
      </div>
    </form>
  );
}
```

What's happening:

- `<textarea>` - Like an input, but for multi-line text
- `rows={4}` - Start with 4 visible rows (users can type more)
- Notice we use `{4}` with curly braces because it's a number, not a string
- Same pattern: label with `htmlFor`, textarea with matching `id`

Step 6: Adding the Giver Input Field

Now let's add a field for who's giving the kudos.

UPDATE: Add the giver field in `src/components/KudosForm.tsx`:

```
export function KudosForm({ onSubmit }: KudosFormProps) {
  return (
    <form className="kudos-form">
      <h2 className="form-title">🎉 Give Kudos</h2>

      <div className="form-group">
```

```
        <label htmlFor="recipient">To:</label>
        <input
          id="recipient"
          type="text"
          placeholder="Enter recipient name"
          required
        />
      </div>

      <div className="form-group">
        <label htmlFor="message">Message:</label>
        <textarea
          id="message"
          placeholder="Write your kudos or feedback..."
          rows={4}
          required
        />
      </div>

      <div className="form-group">
        <label htmlFor="giver">From:</label>
        <input
          id="giver"
          type="text"
          placeholder="Your name"
          required
        />
      </div>
    </form>
  );
}
```

What's happening:

- Same pattern as the recipient field
- This captures who is giving the kudos
- Notice the consistent structure: div wrapper, label, input

Step 7: Adding the Type Selector (Dropdown)

Let's add a dropdown to choose between "kudos" and "feedback".

UPDATE: Add the type selector in `src/components/KudosForm.tsx`:

```
export function KudosForm({ onSubmit }: KudosFormProps) {
  return (
    <form className="kudos-form">
      <h2 className="form-title">🎉 Give Kudos</h2>

      <div className="form-group">
        <label htmlFor="recipient">To:</label>
        <input
          id="recipient"
          type="text"
          placeholder="Enter recipient name"
          required
        />
      </div>

      <div className="form-group">
        <label htmlFor="message">Message:</label>
        <textarea
          id="message"
          placeholder="Write your kudos or feedback..."
          rows={4}
          required
        />
      </div>

      <div className="form-group">
        <label htmlFor="giver">From:</label>
        <input
          id="giver"
          type="text"
          placeholder="Your name"
          required
```

```
        />
      </div>

      <div className="form-group">
        <label htmlFor="type">Type: </label>
        <select id="type">
          <option value="kudos">Kudos</option>
          <option value="feedback">Feedback</option>
        </select>
      </div>
    </form>
  );
}
```

What's happening:

- `<select>` - Creates a dropdown menu
- `<option value="kudos">` - Each option has a value (what gets stored) and text (what user sees)
- The first option (Kudos) is selected by default
- This will let users choose whether they're giving kudos or feedback through the simple logic we impleemnted a few steps ago

---

Step 8: Adding the Submit Button

Finally, let's add the submit button to complete our form structure.

UPDATE: Add the button at the end of the form in `src/components/KudosForm.tsx`:

```
export function KudosForm({ onSubmit }: KudosFormProps) {
  return (
    <form className="kudos-form">
      <h2 className="form-title">□ Give Kudos</h2>

      <div className="form-group">
        <label htmlFor="recipient">To: </label>
        <input
          id="recipient"
          type="text"
          placeholder="Enter recipient name"
```

```
          required
        />
      </div>

      <div className="form-group">
        <label htmlFor="message">Message:</label>
        <textarea
          id="message"
          placeholder="Write your kudos or feedback..."
          rows={4}
          required
        />
      </div>

      <div className="form-group">
        <label htmlFor="giver">From:</label>
        <input
          id="giver"
          type="text"
          placeholder="Your name"
          required
        />
      </div>

      <div className="form-group">
        <label htmlFor="type">Type:</label>
        <select id="type">
          <option value="kudos">Kudos</option>
          <option value="feedback">Feedback</option>
        </select>
      </div>

      <button type="submit" className="submit-button">
        Send Kudos □□
      </button>
    </form>
  );
}
```

What's happening:

- `<button type="submit">` - When clicked, this will submit the form
- `type="submit"` is crucial - it tells the browser this button submits the form
- `className="submit-button"` for styling
- We're doing great work!

Step 8a: Import the Form Component

Now let's see our form structure! Even though it's not styled yet, we can display it to make sure everything is working.

UPDATE: Add imports at the top of `src/App.tsx`:

```
import { useState } from 'react';
import { KudosCard } from './components/KudosCard';
import { KudosForm } from './components/KudosForm';  // ← Add this
import './components/KudosCard.css';
import './App.css';
```

What's happening:

- We import the `KudosForm` component we just created
- Now we can use `<KudosForm />` in our JSX
- Note: We're NOT importing the CSS yet - we'll add that after we create it

---

Step 8b: Create a Temporary Handler Function

Our form expects an `onSubmit` prop (remember the interface we created?). Let's create a temporary function to pass to it.

UPDATE: Add this inside the `App` function in `src/App.tsx`, right before the `return` statement:

```
function App() {
  const sampleKudos = [
    {
      recipient: "Jane Smith",
      message: "Amazing work on the authentication refactor!  Your attention to detail made the
whole team more productive.",
      giver: "John Doe",
      type: "kudos" as const,
      date: "Mar 22, 2024"
    },
    {
```

```
      recipient: "Bob Wilson",
      message: "Could improve code documentation in the API module.",
      giver: "Alice Chen",
      type: "feedback" as const,
      date: "Mar 20, 2024"
    }
  ];

  // Temporary function - we'll make this work properly later!
  const handleAddKudos = (kudos: any) => {
    console.log('New kudos submitted:', kudos);
  };

  return (
    // ... rest of code ...
  );
}
```

What's happening:

- We create a function called `handleAddKudos` that takes a kudos object
- For now, it just logs to the console - we'll make it actually add kudos later
- `kudos: any` - TypeScript type `any` means "any type" (we'll make this properly typed later)
- This function will be called when the form is submitted

Step 8c: Add the Form to the JSX

UPDATE: Add the form component in `src/App.tsx`:

```
return (
  <div className="app-container">
    <div className="app-content">
      <h1 className="app-title">□□ Kudos Board</h1>

      {/* Add the form here! */}
      <KudosForm onSubmit={handleAddKudos} />

      <div className="cards-grid">
        {sampleKudos.map((kudos, index) => (
          <KudosCard
```

```
            key={index}
            recipient={kudos.recipient}
            message={kudos.message}
            giver={kudos.giver}
            type={kudos.type}
            date={kudos.date}
          />
        ))}
      </div>
    </div>
  </div>
);
```

What's happening:

- `<KudosForm onSubmit={handleAddKudos} />` - We render the form component
- `onSubmit={handleAddKudos}` - We pass our temporary function as a prop
- The form appears ABOVE the kudos cards
- The form will receive the `handleAddKudos` function through its props

Step 8d: Check Your Browser!

Look at your browser now!

You should see your unstyled form appear above the kudos cards. It won't look pretty yet (that's coming in Part 3!), but you should see:

- The "Give Kudos" header
- The "To:" label and input field
- The "Message:" label and textarea
- The "From:" label and input field
- The "Type:" label and dropdown
- The "Send Kudos" button
- All the existing kudos cards below

Try clicking around:

- You can type in the fields
- You can select from the dropdown
- If you click the button, the form will submit (though nothing will happen yet - we haven't added the submit logic)

Don't worry that it looks plain! In the next section (Part 3), we'll make it beautiful with CSS!

Revision #18
Created 8 October 2025 05:22:23 by Emilio Cardillo Schrader
Updated 28 October 2025 00:52:01 by Emilio Cardillo Schrader