Assignment 8: React Events, Lifecycle Methods, and Forms

In this assignment we'll continue using the Dog API (combined with an API that you will create yourself) to create a more complex React application. The application will allow the user to select a dog breed from a list and see images of that type of dog with comments from users. The user can also submit comments on the chosen breed and choose how many images are displayed.

In service of this, we'll learn about event handlers in React, lifecycle methods to manage how and when changes occur in components, and controlled forms.

See the **demo** video in the starter files for a demonstration of how the app should work.

I've provided a stylesheet called **style-sample.css** that will arrange your page in a slightly more aesthetically-palatable format. If you structure your HTML and classes differently than I have, this might not work, so don't worry if your results don't look exactly like the demo video.

Task 1: Setup

I've provided a project folder in the starter files that has everything you need to serve a basic API that sends and receives short text comments. (You will see that it has been very lazily-coded and is not a great example of a robust server-side application.) You will have to do a few things to set it up:

- run the command **npm install** in the terminal to install the dependencies;
- create a new database and user in MongoDB Atlas, both called **a8**.

You'll notice that the file **db/connection.js** that I've given you has a bit of a different format than you may be used to. That's because instead of pasting the database connection string directly into our code as we have done in the past, we're going to set the connection string as an **environment variable**. This will allow us to separate our source code from confidential credentials that are needed to make it work.

Create a new file in the root of your project called **.env**. In this file, add the following:

DB_CONNECTION=your-connection-string-here

Obviously, add your own database connection string above. Remember to integrate your new password and the name of the database into the connection string. (Note that string quotes are not needed.)

We've now created a new environment variable called **DB_CONNECTION**. One could add additional credentials and application settings to the **.env** file simply by adding additional lines with new variable **name=value** pairs.

Since this file contains potentially sensitive information, we would normally **not commit this file to a public repository**. Committing API keys, connections strings, etc. to a public repository is an unfortunate way to get hacked!

In order to actually use this environment variable in your application, you'll have to install the **dotenv** package (using NPM). In **db/connection.js**, I've done the following to access the new environment variable:

- Added the following code:
  require('dotenv').config();

  Now we can access your environment variable like this: **process.env.DB_CONNECTION**

- So now we can set the connection string like this :

  let mongoDB = process.env.DB_CONNECTION;

Once you have the database connection working with your new environment variable, you can start the server using **npm run dev** and build the React project using **npm run watch**.

Inspect the code in **app.js**. As you can see the API is serving two routes:

- **GET /comments/:dog,** which gets an array of comments about a specific breed of dog, and
- **POST /comments** which allows one to create a new comment document and save it to the database. Each comment record has the breed of dog that the comment refers to and the comment text.

Additionally, I've set up a **public** and **src** folder with Webpack. Now all you have to do is write the React application!

Task 2: Basic Structure of React Application

The application will consist of an **App** component that contains two sub-components:
- **Chooser**, which shows a list of dog breeds as buttons that the user can select;
- **Details**, which shows images and comments about the chosen breed.

(Feel free to choose better names for your components, but note that you may have to update the **style-sample.css** file I've provided with any new class names you come up with.)

To start, create the **App** component with the following:

- Has a constructor that sets the state with two properties:
    - **dogs:** an array (initialized to []) that will contain the names of all dog breeds;
    - **current:** a string (initialized to undefined) with the currently-chosen dog.
- A **componentDidMount()** lifecycle method that uses Axios to fetch all dog breeds from Dog API, get their names, and store the names in the state of the component. The **current** property in the component state should be set to the first dog in the array.(Remember to use **this.setState** to set the state. Also remember that **componentDidMount()** runs after the component has been successfully created, and is often used to load initial data from an API.)
- A **render()** method that uses JSX to render an **h1** heading, a **Chooser** component, and a **Details** component. It should pass, as **props**, the **current** dog to the **Details** component and the **dogs** array to the **Chooser** component. These should be wrapped in a **div** with a class called **app**. (Remember to use the **className** attribute to set this.)

This **App** component will do more later, but that's enough for now.

Now create the **Chooser** component in a file called **Chooser.js.**

This component is pretty simple; it shouldn't have a state of its own; it should just render an **h2** heading and an **unordered list** that contains button elements with the name of each breed--everything should both be wrapped in a **div** with class **chooser**. For the keys in the list, just use the name of the dog breed, since they're all unique.

Create the **Details** component now. This one will actually contain a lot of stuff, and we'll eventually break it into several smaller components. But for now, just have it render a **div** with class **details** that contains an **h2** heading. The heading should contain the name of the currently-selected dog breed.

Task 3: Choose a Dog Breed

Note: the following documentation will be helpful for this section:
https://reactjs.org/docs/handling-events.html

So, we have a basic structure for allowing the user to choose a dog from the **Chooser** component, by clicking on it, and have the name of that dog appear in the **Details** component. How do we connect these components though? The answer is through the **state** of their common parent, the **App** component.

You might think that we would create an event handler in the **Chooser** component to implement this. However, we're actually going to create an event handler in the parent **App** component and have it attached to an event listener in the **Chooser** component.

In **App.js**, create another method (between **componentDidMount()** and **render()**) with this code:

```
chooseDog(dog) {

  this.setState({
    current:dog
  })
}
```

As you can see, the method takes a dog as an argument and updates the state with that dog as **current**. We're eventually going to use this method as an event handler for the buttons in the Chooser component!

In order for this to work, we need to make sure that the **this** that is referred to in **chooseDog** is the right **this.** That is, the **this** of the **App** component! (Sorry, it's a bit confusing!) When we call **this.setState** in **chooseDog**, it has to be the state of **App** that we're setting. We can use the **bind** function to do this.

In the **constructor** of **App.js**, add the following code:

```
this.chooseDog = this.chooseDog.bind(this); // refers to this of App
```

The above code effectively ensures that within the **chooseDog** method, **this.setState** will set the state of the App component.

Now, we need some way for the buttons in **Chooser** to call this function. To do this, pass the **chooseDog** method to **Chooser** as a prop; your JSX code should look like this:

<Chooser dogs={this.state.dogs} chooseDog={this.chooseDog} />

So now within the **Chooser** component, you have access to this method via **this.props.chooseDog.**

React uses so-called **synthetic** events rather than the normal events that you're used to. However, they work almost exactly the same way. Event names use camelCase, so **onclick** becomes **onClick**, **onsubmit** becomes **onSubmit**, etc.

Place the following code inside the **<button>** element in the render method of **Chooser**:

onClick={()=>this.props.chooseDog(dog)}

As you can see, the **onClick** event above is being assigned a function as a value. That function simply calls the **chooseDog** function with the name associated with the current button. (note that you will have to change **dog** to whichever variable is storing the current dog within your code. Copying and pasting the above code without making this change will not work.) Since we took special care to make sure **chooseDog** updates the state of the **App** component, the user should now be able to change the currently-selected in **App** dog by clicking on the buttons in Chooser! You should see the heading in the **Details** component being updated accordingly.

We've finally created a React app that has some interactivity! We're pretty much done with the **Chooser** component. Let's focus on the **Details** component now.

Task 3: Get Images for Chosen Breed

Let's extend the **Details** component by adding a **constructor** that keeps track of the **state** of the component. For now, just have two properties in the state:

- **images:** an array (initialized to empty) with the URLs of all images of dogs of this breed;
- **number:** the number of images to show on the page, initialized to **1**.

Check the Dog API https://dog.ceo/dog-api/ documentation to see which endpoint can be used to get all the images for a certain breed of dog.

You might be tempted to use the **componentDidMount()** method to load all images using the **current** dog (passed down from **App** as a **prop**) with an Axios request. However, the problem is that when this component has mounted, the Ajax request happening in the **App** component that fetches the dog breeds may not have completed. So the **current** dog may be **undefined** when this component mounts! Not only that, but the user can click on another dog at any time to change the **current** dog. And since the **Details** component will have already finished mounting by that time, we won't get the updated images for the newly-chosen dog!

The solution here is to use a different lifecycle method called **componentDidUpdate**. This method is called after component **props** and **state** have been updated, which makes it the right choice for fetching updated images via Axios when the **current** dog prop changes.
Note that while we're only using the lifecycle methods **componentDidMount, componentDidUpdate**, and **render(),** there are a few others available to us. You can check them out here:
https://reactjs.org/docs/react-component.html
or here for a more focused overview:
https://blog.bitsrc.io/react-16-lifecycle-methods-how-and-when-to-use-them-f4ad31fb2282

Warning: a few years ago, React lifecycle methods received a significant update, and a bunch of old methods were deprecated. Therefore, there's a lot of out-of-date and wrong information on the web about this subject! I would advise not reading any articles about React lifecycle from earlier than 2018. Unfortunately, this includes the textbook we're using. However, the above articles provide good, current info.

In **Details.js**, add the following method:

```
componentDidUpdate(prevProps, prevState) {

    // Load new updated images here
}
```

The **prevProps** parameter contains the old **props**, before they were updated. (it's the same for **prevState**, although we won't use that here.) This is useful because it allows you to check to see if the new props are the same as the old props.

In our case, we would like to check to see if the **current** dog breed value has changed. If it has, we should use an Axios request to the Dog API to get an array of images for the **current** breed. Figure out how to do this. When you get the array, update the **images** state to store this data.

Now modify the **render** method in **Details** to render the images! Use an appropriate key for each item in the list. (you don't have to use an actual HTML list, but each **<img>** tag still has to have a key.)

You can start by rendering all images of the chosen breed. Render them all into a **div** with the class **'images'.** However, remember that we've created a state property that keeps track of the **number** of images to render. So, modify your code to render *only this many images*. Hint: use the **slice** function to create a new array that is a subset of the **images** array:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice

Task 4: Intro to Forms

Forms in React work a bit differently than what you may be used to; normally, the state of the form (including the data the user has entered) is kept by the DOM in the browser. However, the recommended method for implementing forms in React is to have the form reflect the state of the component, and update the state dynamically as the data in the form is changed.

This will allow us to implement a form that actually dynamically updates the application as the user interacts with it! We're going to use this to implement a form that allows the user to choose how many images to display. They won't even have to submit the form since the changes will happen on the fly.

In the render method in the **Details** component, place the following JSX code:

```
<form>
        <label>
         number of images:
            <input type='number' min='0' max={ this.state.images.length}
            value={this.state.number} onChange={this.handleNumberChange}/>
        </label>
</form>
```

There's a lot going on here, so let's break it down:

- We've set up a form with a single input that allows the user to choose a number. There is no submit button since the form will update the component in real time as the user changes the number;
- The maximum number of images has been set to the number of **images** contained in the state of the component (using the **max** attribute);
- The **value** of the input (which corresponds to the data currently entered into it) is set to the current **number** state of the component;
- We've attached an event listener for **change** events (that happen when the user changes the data in the input).

Your task is to implement the **handleNumberChange** method. Start with this code:

```
handleNumberChange(event) {
    // TODO update the state to reflect the new number
  }
```

Hint: this resource will be helpful: https://reactjs.org/docs/forms.html

Once you have this working, you should be able to rapidly change the number of images displayed on the page!

Task 5: Intermediate Forms

Our next task is to implement the form that allows the user to submit a comment for the currently-selected dog. Remember that I provided the server-side code for the API that will handle the POST request with the new comment data.

To start, we should add a few more properties to the **state** of the **Details** component:

- **newComment**: a string (initialized to '') that keeps track of the comment the user is typing into the form. It should be updated every time the user types a new character.
- **comments**: an array (initialized to []) that contains a list of objects representing the comments retrieved from the server-side API.

In **Details.js**, place the following JSX code in the **render()** function to display the form:

```
<form onSubmit={this.handleCommentSubmit}>
            Submit a comment:
                <textarea onChange={this.handleCommentChange}
                value={this.state.newComment}></textarea>
            <input type='submit' value='Submit comment' />
 </form>
```

As you can see here, we're keeping the **<textarea>** input synchronized with the **state** of the **Details** component via the **value** and **onChange** attributes. However, we also have a submit button that invokes the **onSubmit** event for the form. We need this so that we can initiate the Axios post request to submit the comment to the server. (this is less dynamic than the previous form.)

Your task is to implement the following two methods in Details:

```
handleCommentChange(event) {

        // Update newComment in state
  }

  handleCommentSubmit(event) {
    event.preventDefault();

        // Post a new comment to '/comments'
  }
```

A few things to remember:

- On the sever side, the **DogComment** model has two fields: **breed** (which has the name of the breed) and **text** (which has the text of the comment).

- When the comment has been posted, the state of the **Details** component should be updated to reset **newComment** to an **empty string** and add the new comment that was just created to the **comments** state.
    - Remember that you can't mutate the state! I recommend getting a clone of the current **comments** array by using **slice**, pushing the new comment into the cloned array, and then using it to set the state.

You now should have the capability to add comments for a current dog breed and have those comments reflected in the state of **Details**!

Update the render method in **Details** to show a list of comments. (Us an actual HTML unordered list for this inside a **div** with the class **'comments'**.

One other thing we need to think about is loading all the comments into the state when we select a different **current** dog breed in the **App** component.

Figure out how to change the **componentDidUpdate** method to load the comments when choosing a different **current** dog.

Task 6: Re-evaluating the Component Structure

By now, you've probably noticed that the **render()** method in **Details** is getting a bit crowded. We should probably break up some of the content we're rendering here into different components that will be nested inside the **Details** component. There are a few ways to do this, but to keep things simple, create two new components: (in my solution, I called them **DetailForms** and **DetailViews**, but feel free to choose different names.)

- **DetailForms:** contains the two forms;
- **DetailViews:** contains the list of images and list of comments.

You could break these components down even further if you wanted to.
Neither of the above components needs to keep its own state; they can simply receive everything they need to render as **props**. However, you're going to find that an annoyingly-large number of props needs to be passed to each of these two components.

If you want, you can just pass them the way you normally would: by adding a bunch of prop attributes to the JSX component tags. However, another way you could do it is by creating an object that contains all the props you need and spreading it into the component tag, like this:

```
render() {

let formProps = {
        handleCommentSubmit:this.handleCommentSubmit,
        handleCommentChange:this.handleCommentChange,
        handleNumberChange:this.handleNumberChange,
        images:this.state.images,
        number:this.state.number,
        newComment:this.state.newComment
      };

       return <DetailForms {...formProps} />; // I removed the other stuff for clarity
}
```

You might find that the above is a bit more readable. Sometimes, you may get lucky and not even have to create a new object to spread in; you could just do this:

```
<SomeComponent {...this.props} />
```

(Unfortunately, that doesn't apply here.)

Hand In

Delete **node_modules, .c9.** Download your project from Cloud 9, extract it, rename the folder to **a8_firstname_lastname**, zip it again, and hand it in to D2L.

Checklist

- [3] App component with state; loads initial data and renders subcomponents;
- [2] Chooser component with list of dogs;
- [2] Event handler for choosing current dog;
- [2] Details component renders number of images as recorded in state; updates when new dog chosen;
- [2] Form for changing number of images;
- [2] Form for submitting new comment
- [2] Details component shows correct comments when comment is submitted or dog changed;
- [1] Details component decomposed appropriately with spread props;

Total: 16
Note that up to -3 may be deducted for improper hand in, poorly-formatted code, etc.