Bootcamp Capstone Project

Image Generator

Part 01 App Creation

1. Use the bare package, install it and then. This zipped file also has an assets folder with images in it. We will use this folder later in the project.

2. Create a folder called styles in your src folder

3. Inside the styles folder create a css file called ImageGenerator.module.css

4. insert this style:

.heading {

font-size: 24px;

margin-bottom: 20px;

color: #333;

}

5. Create a folder called components inside the src folder then create a Header.js file there

6. Insert the following lines of code:

import React from "react";

import styles from "./../styles/ImageGenerator.module.css";

const Header = () => <h1 className={styles.heading}>AI Image Generator</h1>;

export default Header;

Note: it is not necessary to import React anymore, but it won’t hurt to have it in the code.

7. In App.js add these two lines before the *function* keyword:

import React from "react";

import Header from "./components/Header";

8. For the return statement in app.js (remove the original line between the div tags):

return (

<Header />

)

9. The next component we need will be the main one.

it will be the most important file, lets call it **ImageGenerator**

Create this component by first adding a file in the components folder and adding a function to be exported, so create ImageGenerator.js

10. Once you have the raw js file, add these two lines

import React, { useState } from "react";

import Header from "./Header";

Inside of the ImageGenerator.js file useState is a hook that will keep track of React variables/properties

11. In the ImageGenerator.js file return the Header component

|  |
| --- |
| import React, { useState } from "react";  import Header from "./Header";  //  function ImageGenerator() {  return (  <>  <Header />  </>  );  }; export default ImageGenerator |

Notice that we imported *Header* but not using it as yet. It has to be removed from App.js and in that file we import image generator instead

|  |  |
| --- | --- |
| File structure at the end of part 1  A screenshot of a computer  Description automatically generated | Output at end of Part 1  A screenshot of a computer  Description automatically generated |

-------------------- end of part 1 -----------------------------------------

Part 02 Adding the Form to Collect the Prompt

1. Add a .js file in the components folder to handle the users prompt, call it InputForm.js
2. Import the same packagaes like for the ImageGenerator file, create a function called InputForm and export that function

|  |
| --- |
| import { useState } from "react";  function InputForm() {  return (  <form>  </form>  );  };  export default InputForm; |

1. Add the following CSS to the ImageGenerator.module.css file:

|  |
| --- |
| .form {  display: flex;  flex-direction: column;  align-items: center;  width: 100%;  }    .input {  width: 100%;  padding: 12px 15px;  margin-bottom: 25px;  border: 2px solid #dcdcdc;  border-radius: 6px;  font-size: 18px;  }  .input:focus {  border-color: #007bff;  box-shadow: 0 0 8px rgba(0, 123, 255, 0.2);  outline: none;  }  .button {  padding: 12px 24px;  background-color: #007bff;  color: white;  border: none;  border-radius: 6px;  cursor: pointer;  font-size: 16px;  }  .button:hover {  background-color: #0056b3;  transform: translateY(-2px);  box-shadow: 0 4px 8px rgba(0, 123, 255, 0.3);  }  .button:disabled {  background-color: #cccccc;  color: #666666;  cursor: not-allowed;  }  .button:disabled:hover {  transform: none;  box-shadow: none;  } |

1. Utilize a style for the form, but import the style file first:

|  |
| --- |
| import React, { useState } from "react";  import styles from "./../styles/ImageGenerator.module.css";  function InputForm() { |

1. Add an input box :

|  |
| --- |
| return (  <form>  <input  className={styles.input}  type="text"  value=""  placeholder="Type your prompt here..."  /> |

1. Add a button to submit the user prompt:

|  |
| --- |
| <button className={styles.button} type="submit">  Generate Image  </button>  </form>  ); |

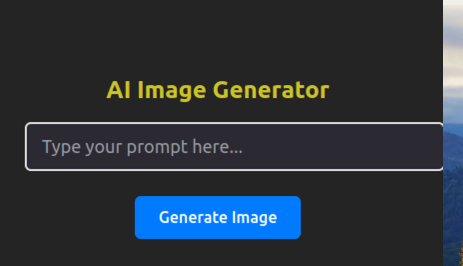
1. Now add the form to the Image generator js file just like we did for Header

|  |
| --- |
| import Header from './components/Header';  import InputForm from './InputForm';  import './App.css'  //  function App() {  const [count, setCount] = useState(0)  return (  <>  <Header />  <InputForm />  </>  ) |

1. We will need useState in this component so add it now:

|  |
| --- |
| import { useState } from "react";  import Header from "./Header";  import InputForm from './InputForm |

1. Replace the Header component with ImageGenerator in the App.js file.



--------------------------- end of part 2 --------------------------------------------

Part 3

In part 3 we will add a grid to show our generated images. However we need an *ImageComponent* first. This is the file that will make the request to our back-end to get the image based on our prompt.

1. This component requires its own style sheet as we will be using quite a few styles and its better for organization. Create a new file called ImageComponent.js inside of the components folder

|  |
| --- |
| A diagram of a software development  Description automatically generated |

1. Create a new .css file called ImageComponent.module.css. For now, add a style for the image:

|  |
| --- |
| .image {  width: 100%;  height: 100%;  border-radius: 12px;  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);  } |

1. Inside the ImageComponent.js file, add the usual *useState* but we will also need *useEffect* in here:

|  |
| --- |
| import { useState, useEffect } from "react";  import styles from "../styles/ImageComponent.module.css"; |

1. Add the skeleton of the component:

|  |
| --- |
| function ImageComponent() {  return (  <div>  </div>  )  };  export default ImageComponent; |

1. This component will be using a service from a utility file. This utility file will do just one thing, pass our prompt to an external service to generate our image. Create a folder called utils in your src folder, and in there create a file api.js:

|  |
| --- |
| export const generateImage = async (prompt) => {  const url = `https://image.pollinations.ai/prompt/${prompt}`;  const response = await fetch(url);  if (!response.ok) {  throw new Error("Error generating image");  };  }; |

The code in #5 shows that we receive a string which we assign to prompt. We can then pass that prompt over to the url service that will generate an image based on the request in the prompt. The prompt string itself will originat in our InputForm.

1. For this particular service, image.pollinations.ai, we need to pass a random number over to it called a *seed*. We can easily get this from the random function, just multiply by 1000:

|  |
| --- |
| export const generateImage = async (prompt) => {  const seed = Math.floor(Math.random() \* 1000);  const url = `https://image.pollinations.ai/prompt/${prompt}`; |

1. We should also verify the prompt using a built-in function called *encodeURIComponent*. According to the documentation this function will encode a text string as a **valid** Uniform Resource Identifier (URI)

|  |
| --- |
| export const generateImage = async (prompt) => {  const encodedPrompt = encodeURIComponent(prompt);  const seed = Math.floor(Math.random() \* 1000);  const url = `https://image.pollinations.ai/prompt/${ encodedPrompt}`; |

Once you have this new variable, use it in the url const. For a note on encodeURIComponent see Appendix B.

1. Just like we had to do a second *await* or *then* for JSON data, we need to this same procedure for *blob* data:

|  |
| --- |
| const response = await fetch(url);  if (!response.ok) {  throw new Error("Error generating image");  };  const blob = await response.blob(); |

Refer to Appendix B for an explanation of the BLOB type

1. Since we are dealing with images here, we should expect to receive a *blob type* from our request. We need to now convert that blob object to a proper URL for displaying purposes. There is a static method on the URL object that can be used for this, its called *createObjectURL*:

|  |
| --- |
| if (!response.ok) {  throw new Error("Error generating image");  };  const blob = await response.blob();  return URL.createObjectURL(blob); |

1. One final task here is to add all the requirements in our request URL for this service, including the seed from #7:

|  |
| --- |
| const encodedPrompt = encodeURIComponent(prompt);  const url = `https://image.pollinations.ai/prompt/${encodedPrompt}? model=flux&width=1280&height=720&seed=${seed}&nologo=true&enhance=true`;  const response = await fetch(url); |

A diagram of a software development

Description automatically generated

Part 4

1. Back in the ImageComponent.js file, import the generateImage() function from the api.js file in the utils folder:

|  |
| --- |
| import styles from "../styles/ImageComponent.module.css";  import {generateImage} from "./../utils/api";  function ImageComponent() { |

You created this function in part 03 #5 and exported it.

1. Add the following style to display our images in a grid. Do this in the ImageGenerator.module.css file:

|  |
| --- |
| .images {  display: grid;  grid-template-columns: repeat(2, 1fr);  gap: 20px;  margin-top: 30px;  width: 100%;  } |

At the moment we will only display one image, but later we will have four or how many you wish to generate.

1. Just like for the form, we will create a new .js file called ImageGrid.js in the components folder

|  |  |
| --- | --- |
|  | This image shows the folder structure at this point in the development. |

1. As usual with new files import the necessary support files and css. Also import the *ImageComponent* you created in part 03:

|  |
| --- |
| import styles from "./../styles/ImageGenerator.module.css";  import ImageComponent from "./ImageComponent"; |

1. Add a function and return just an empty pair of div tags for now:

|  |
| --- |
| function ImageGrid(){  return (  <div></div>  )  };  export default ImageGrid; |

1. Before we wrap up this part, add a new utility file called placeholders.js and export an array of placeholder images:

|  |
| --- |
| import placeholder1 from "../assets/image1.jpg";  import placeholder2 from "../assets/image2.jpg";  import placeholder3 from "../assets/image3.jpg";  import placeholder4 from "../assets/image4.jpg";  export const placeholderImages = [  placeholder1,  placeholder2,  placeholder3,  placeholder4,  ]; |

These images are in the assets folder. So at this point, you should have two files in your utils folder, api.js and placeholder.js which will have the lines in #6 and nothing else.

--------------------------- end of part 4 --------------------------------------------

Part 5

In part 5 we will continue to develop the ImageComponent.js file.

1. We will need a *placeholder image* and a *prompt* to generate a new image, so add these two parameters to our function:

|  |
| --- |
| function ImageComponent( {placeholder, prompt} ) { |

Remember to use de-structuring syntax like we did for *PhoneNumber* in the **Intro to Hookz** exercise. This component will be called from ImageGrid, but ImageGrid will have several attributes, hence the need for de-structuring here. So the values for both of these parameters will be passed in from ImageGrid. (Refer to Part 7 #1 to see how these are passed)

Inside the ImageComponent function, add a const to store an image as well as a function to change that image:

|  |
| --- |
| function ImageComponent(placeholder, prompt) {  const [image, setImage] = useState(placeholder); |

We will get an image from the fetchImage() function below but initially we get it from placeholder.

1. Now add a function that will eventually call generateImage() function in the utils/api file:

|  |
| --- |
| const [image, setImage] = useState(placeholder);  async function fetchImage(){  try{  const newImage = await generateImage(prompt);  setImage(newImage);  } catch(err){  //  } finally {  }  }; |

Notice that we also store the image once it comes back from the *util* file, essentially replacing the placeholder image (see #3 below).

1. Display the image received or from the placeholder between the <div> tags:

|  |
| --- |
| return (  <div>  <img src={image} alt="Generated" className={styles.image} />  <button>Refetch</button>  </div> |

So, from #1 above, when we created the state variable image, we populate that variable with whatever we get from the placeholder parameter. Remember this comes from ImageGrid. So initially that image gets shown between the <div> tags but will eventually get replaced with the AI generated one.

1. Now on the ImageGrid.js file, import the placeholder images from the placeholder utility:

|  |
| --- |
| import ImageComponent from "./ImageComponent";  import { placeholderImages } from "./../utils/placeholder";  function ImageGrid() { |

1. Then in the return statement, return the ImageComponent:

|  |
| --- |
| <div className={styles.images}>  <ImageComponent>  </ImageComponent>  </div> |

1. In the ImageGenerator file, import the InputForm and the ImageGrid:

|  |
| --- |
| import { useState } from "react";  import ImageGrid from "./ImageGrid";  import InputForm from "./InputForm";  import Header from "./Header"; |

1. Add both components to the return statement of ImageGenerator:

|  |
| --- |
| function ImageGenerator() {  return (  <>  <Header />  <InputForm />  <ImageGrid />  </> |

Part 6

In part 6 we will pass the prompt the user enters over to the api code. The prompt starts with the Input Form and then makes it to the api in the utils folder.

1. We will need a state variable in the InputForm to hold the prompt that the user enters and a function to handle that prompt:

|  |
| --- |
| import { useState } from "react";  import styles from "./../styles/ImageGenerator.module.css";  function InputForm({ onGenerate, isGenerating } ) { |

1. When the InputForm.js file is called from ImageGenerator.js file, two items get passed. One is a pointer to a function that starts the image generation process and the other is Boolean state value that keeps track of whether the image generating process is active or complete. In the InputForm.js file, we need to de-structure those two pointers via the function’s parameters:

|  |
| --- |
| import { useState } from "react";  import styles from "./../styles/ImageGenerator.module.css";  function InputForm( { onGenerate, isGenerating } ) { |

The property onGenerate is a function and isGenerating is a booean value. Both of these pointers will get passed when the IputForm.js file is called from ImageGenerator.js file.

1. Now add the state variable at the top of the InputForm.js file, this is for the user’s prompt:

|  |
| --- |
| function InputForm() {  const [inputPrompt, setInputPrompt] = useState("");  return ( |

1. Just below the state variable, add a function to handle the event of the user entering a value into the form field:

|  |
| --- |
| const [inputPrompt, setInputPrompt] = useState("");  const handleInputChange = (e) => {  setInputPrompt(e.target.value);  };  return ( |

So, once the new value is entered, that value gets stored in the *inputPrompt* property

1. In the ImageGenerator file, pass these two pointers:

|  |
| --- |
| return (  <>  <Header />  <InputForm  onGenerate={handleGenerateClick}  isGenerating={isGenerating}  />  <ImageGrid  prompt={prompt} |

So now when the form is submitted the onGenerate() function is triggered and the handleGenerateClick() funciotn **on the imageGenerator** file is executed. That transfers the new prompt to the **ImageGrid** file. We will write the ImageGenerator functions shortly.

1. Now we can complete the handleSubmit() function inside the InputForm:

|  |
| --- |
| setInputPrompt(e.target.value);  };  const handleSubmit = (e) => {  e.preventDefault();  onGenerate(inputPrompt);  };  return ( |

1. Since we have access to the *isGenerating* property, lets use that value to either enable or disable the “Generate Image” button:

|  |
| --- |
| onChange={handleInputChange}  />  <button className={styles.button} type="submit" disabled={isGenerating}>  Generate Image  </button>  </form> |

1. At the same time, we can use this property to display an appropriate message on the button itself:

|  |
| --- |
| onChange={handleInputChange}  />  <button className={styles.button} type="submit" disabled={isGenerating}>  {isGenerating ? "Crafting your masterpiece..." : "Generate Images"}  </button>  </form> |

1. Add code to handle the form submission. We will also add a style to the form:

|  |
| --- |
| return (  <form onSubmit={handleSubmit} className={styles.form} >  <input |

1. To prevent errors at this point, add an empty function in the ImageGenerator file:

|  |
| --- |
| function ImageGenerator() {  const handleGenerateClick = (newPrompt) => {  };  return ( |

1. Also declare the isGenerating Boolean variable along with it’s setter function:

|  |
| --- |
| function ImageGenerator() {  const [isGenerating, setIsGenerating] = useState(false);  const handleGenerateClick = (newPrompt) => {  }; |

----------------- end of part 6 ----------------------

Part 7

In part 7 we will focus on the request and response. Remember that the *ImageGenerator* is calling both the *ImageGrid* and the *InputForm*. The *ImageGrid* in turn calls the *ImageComponent*. We need to supply the *ImageGrid* with three things, the **prompt** itself, a Boolean value deciding if we are currently **generating** anything and a **function** to call once the generation process is complete.

1. In the ImageGenerator file, we get the prompt from the InputForm. The *isGenerating* variable is initially set to false and we will write the handleGenerationComplete() function in #12 below which will update the isGenerating variable if it changes:

|  |
| --- |
| isGenerating={isGenerating}  />  <ImageGrid  prompt={prompt}  isGenerating={isGenerating}  onGenerationComplete={handleGenerationComplete}  /> |

1. Now in the ImageGrid itself, lets adjust the parameters to receive a de-structured object with the three items, two properties and one function:

|  |
| --- |
| function ImageGrid( { prompt, isGenerating, onGenerationComplete } ) {  return ( |

1. Add these items to the ImageComponent selector from inside the ImageGrid file:

|  |
| --- |
| <ImageComponent  prompt={prompt}  isGenerating={isGenerating}  onGenerationComplete={onGenerationComplete}  >  </ImageComponent> |

1. We can now focus on the ImageComponent.js file. Lets also pass the same three items we passed for ImageGrid into the ImageComponent. We already have placeholder and prompt, we now add isGenerating and onGenerationComplete:

|  |
| --- |
| function ImageComponent(  {placeholder, prompt, isGenerating, onGenerationComplete }  ) {  const [image, setImage] = useState(placeholder) |

1. Add two more lines of state variables, one for isLoading and one for any errors that might occur:

|  |
| --- |
| {placeholder, prompt, isGenerating, onGenerationComplete}  ) {  const [image, setImage] = useState(placeholder);  const [isLoading, setIsLoading] = useState(false);  const [error, setError] = useState(null);  async function fetchImage() { |

The ImageComponent will be the first to know if an image was generated or if any errors occurred.

1. In part 3 we imported useEffect into the ImageComponent file. Lets now use it to fire the fetchImage() function in the api.js file. That will interface with our API to get the new image:

|  |
| --- |
| };  useEffect(() => {  if (isGenerating) {  fetchImage();  }  }, [isGenerating, fetchImage] );  return ( |

This code will run if isGenerating changes or fetchImage gets called from anywhere else in the code.

1. Now we can complete the catch section of the try…catch block:

|  |
| --- |
| setImage(newImage);  } catch (err) {  setError("Error in generating image: "+ err);  } finally { |

1. Now that we have an isLoading state variable and we have access to the onGenerationComplete() function, we can handle both those cases in the finally block:

|  |
| --- |
| } catch (err) {  setError("Error in generating image: "+ err);  } finally {  setIsLoading(false);  onGenerationComplete();  } |

1. In the ImageGrid add a style to the div and add the *placeholder* property:

|  |
| --- |
| <div className={styles.images}>  <ImageComponent  placeholder={placeholderImages[0]}  prompt={prompt}  isGenerating={isGenerating}  onGenerationComplete={onGenerationComplete}  > |

Remember the placeholder.js returns an array of four images, we will use just one image for now.

1. Back in the ImageGrid file, add a style to the div containing the image(s):

|  |
| --- |
| <div className={styles.images}>  <ImageComponent  placeholder={placeholderImages[0]}  prompt={prompt}  isGenerating={isGenerating}  onGenerationComplete={onGenerationComplete}  > |

Remember the placeholder.js returns an array of four images, we will use just one image for now.

1. In Part06 we added the isGenerating Boolean into the ImageGenerator file, so now lets also add the prompt. This prompt value gets passed to ImageGrid and eventually the api:

|  |
| --- |
| function ImageGenerator() {  const [prompt, setPrompt] = useState("");  const [isGenerating, setIsGenerating] = useState(false);  const handleGenerateClick = (newPrompt) => { |

1. We still need to define the handleGenerationComplete in the ImageGenerator file:

|  |
| --- |
| setIsGenerating(true);  };  const handleGenerationComplete = () => {  setIsGenerating(false);  };  return ( |

1. We also still need to complete the handleGenerateClick() function:

|  |
| --- |
| const [prompt, setPrompt] = useState("");  const handleGenerateClick = (newPrompt) => {  if (!newPrompt || isGenerating) return;  setPrompt(newPrompt);  setIsGenerating(true);  };  const handleGenerationComplete = () => { |

1. When we created the form in Part02 #5, we did not complete the handleInputChange() function, lets do that now in the InputForm. At the same time lets make sure that the value of the input form control is the value of the inputPrompt:

|  |
| --- |
| <form onSubmit={handleSubmit} className={styles.form} >  <input  className={styles.input}  type="text"  value={inputPrompt}  onChange={handleInputChange}  placeholder="Type your prompt here..."  /> |

Part 8

We will continue improving the basic code we have so far. In this part we will add a *refetch* button and improve the fetchImage()function. In the part9 we will add a function to handle caching. Also in part9 we will utilize all four image placeholders and generate four images.

1. In the ImageComponent file, when we display the generated image or images, we should add a refetch button there for the user to perhaps ask for a slightly different image:

|  |
| --- |
| return (  <div>  <img src={image} alt="Generated" className={styles.image} />  <button>  Refetch  </button>  </div> |

1. Add this style for the button, in the ImageComponent.module.css file:

|  |
| --- |
| .refetchButton {  position: absolute;  background-color: rgba(0, 123, 255, 0.8);  color: white;  border: none;  padding: 8px 16px;  border-radius: 4px;  cursor: pointer;  font-size: 14px;  } |

1. Just like the button on the InputForm, we can add an *onClick* event and point it to the fetchImage() function but give it a style first:

|  |
| --- |
| <button  className={styles.refetchButton}  onClick={fetchImage}  >  Refetch  </button> |

1. Also disable the button when the image processing begins:

|  |
| --- |
| <button  className={styles.refetchButton}  onClick={fetchImage}  disabled={isLoading}  >  Refetch  </button> |

1. We probably do not want this refetch button to show up if the page just loaded or user has not yet interacted with the app, so use a short-circuit technique here:

|  |
| --- |
| <img src={image} alt="Generated" className={styles.image} />  {image !== placeholder && (  <button  className={styles.refetchButton}  onClick={fetchImage}  disabled={isLoading}  >  Refetch  </button>  )}  </div> |

1. In ImageComponent we have the *isGenerating* Boolean value that we have not used so far. This can be used to supply a loading message to the user. In this case we have a pulsating image we can show while we wait for the actual images. Add this CSS style in the ImageComponent.module.css file:

|  |
| --- |
| .skeleton {  width: 100%;  height: 100%;  background: linear-gradient(90deg, #f0f0f0 25%, #e0e0e0 50%, #f0f0f0 75%);  background-size: 200% 100%;  animation: pulse 1.5s infinite ease-in-out;  border-radius: 12px;  } |

1. Lets also add an image wrapper class using the following style:

|  |
| --- |
| .imageWrapper {  position: relative;  width: 320px;  height: 320px;  background-color: #f0f0f0;  border-radius: 12px;  display: flex;  align-items: center;  justify-content: center;  overflow: hidden;  } |

1. Add the imageWrapper class to the div that has our refetch button and returned images:

|  |
| --- |
| return (  <div className={styles.imageWrapper} >  <img src={image} alt="Generated" className={styles.image} />  {image !== placeholder && ( |

1. We now have to decide what shows up when the image(s) are being processed and what shows after the images have returned in complete form. We will create a pair of <div> tags and use the skeleton style we added in #7 when the image is being generated. When the image(s) return, show the images and the refetch button. First the isLoading part:

|  |
| --- |
| return (  <div className={styles.imageWrapper}>  {isLoading ? (  <div className={styles.skeleton}></div>  ) : (  <img src={image} alt="Generated" className={styles.image} />  {image !== placeholder && ( |

1. Now to handle the not isLoading part we need to wrap the image and button in fragments.

|  |
| --- |
| <div className={styles.imageWrapper}>  {isLoading ? (  <div className={styles.skeleton}></div>  ) : (  <>  <img src={image} alt="Generated" className={styles.image} />  {image !== placeholder && (  <button  className={styles.refetchButton}  onClick={fetchImage}  disabled={isLoading}  >  Refetch  </button>  )}  </>  </div> |

1. Close off the button and image with a matching closing parenthesis and curly braces to match the opening pair where the isLoading Boolean variable is:

|  |
| --- |
| <div className={styles.imageWrapper}>  {isLoading ? (  <div className={styles.skeleton}></div>  ) : (  <>  <img src={image} alt="Generated" className={styles.image} />  {image !== placeholder && (  <button  className={styles.refetchButton}  onClick={fetchImage}  disabled={isLoading}  >  Refetch  </button>  )}  </>  )}  </div> |

1. There is one more thing we need to deal with here and that is error display. We already handle any errors in the try…catch block but have not shown any error consequences. Just before the ending <div> add an error short circuit line:

|  |
| --- |
| </>  )}  error && <div>{error}</div>;  </div>  )  };  //  export default ImageComponent; |

1. Add a CSS class for the error:

|  |
| --- |
| .error {  color: #ff0000;  font-size: 14px;  margin-top: 8px;  text-align: center;  } |

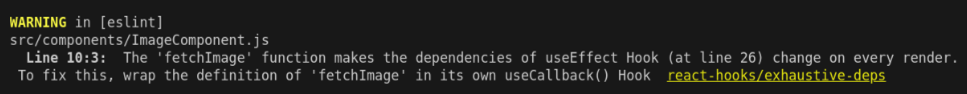
1. Just adjust the error div to apply this style:

|  |
| --- |
| </>  )}  { error && <div className={styles.error} >{error}</div> }  </div>  )  };  export default ImageComponent |

Wrap the entire error block in curly braces to prevent it showing up on the screen as HTML text

Part 9

If you look in the console window you will see a message like this one showing. This is just a warning but we need to take care of it.



The message is actually pointing to the hook we should be using, its the useCallback() hook.

This hook takes a function as the first parameter. That function is then momoized, meaning that it lives in memory and is ready to execute when the funciton is needed.

Usually, functions get created in memory as the component is loaded. This is no different than using the *new* operator to instantiate a class object.

In our case the fetchImage() function is inside of a useEffect() hook, which is the perfect scenario for implementing a useCallback().

We need to wrap the try…catch block of fetchImage() into the useCallback() function. This means that we should re-structure the fetchImage and make it a function expression:

|  |
| --- |
| const fetchImage = async () => {  try{ |

1. Now we can wrap the entire block of code inside of the useCallback() function:

|  |
| --- |
| const fetchImage = useCallback( async () => {  try{  const newImage = await generateImage(prompt);  setImage(newImage);  } catch(err){  setError("Error in generating image: "+ err);  } finally {  setIsLoading(false);  onGenerationComplete();  };  } ) ; |

Remember to import the useCallback module from react at the top of the ImageComponent file.

1. To test the error images and the isLoading feature, you can try to spoil the url being used to generate the image. For example instead of the word prompt, change it to promp:

|  |
| --- |
| const encodedPrompt = encodeURIComponent(prompt);  const url =  `https://image.pollinations.ai/promp/${prompt}?  model=flux&width=1280&height=720&seed=${seed}&nologo=true&enhance=true`;  const response = await fetch(url); |

Remember to change it back to prompt when you leave this file.

Part 10 – generating as much images as are in the placeholder array

In the placeholder.js file there is an array called *placeholderImages*. It contains four placeholder images. Right now in the ImageGrid file, we are only working with one of these images. We now need to handle all four placaholders and generate four images to replace them.

1. We need to generate four ImageComponents based on the four elements in the placeholder array. The ImageGrid will create one component per array image:

|  |
| --- |
| return (  <div className={styles.images}>  { placeholderImages.map( ( placeholder, index ) => ( ) ) }  <ImageComponent |

1. With map() we can iterate over the placeholder array and then call ImageComponent for each placeholder we have. So now just get the ImageComponent code inside of the parenthesis of the map() method:

|  |
| --- |
| <div className={styles.images}>  {placeholderImages.map((placeholder, index) => (  <ImageComponent  placeholder={placeholderImages[0]}  prompt={prompt}  isGenerating={isGenerating}  onGenerationComplete={onGenerationComplete}  >  </ImageComponent>  ))}  </div> |

At this point you should be getting four of the same image, but see #3 below.

1. Notice that the warning about keys are already showing up in the console window. Luckily we have access to the index with the map() method, so just apply it for each ImageComponent:

|  |
| --- |
| <ImageComponent  key={index}  placeholder={placeholder}  prompt={prompt} |

1. How do we know when we have completed the image generation process now that we have four images. Well we could use the index from the map() method for this. When this index gets to the total of the array we are done. At this time we can call the onGenerateComplete() method. We will have to use an inner function for this:

|  |
| --- |
| prompt={prompt}  isGenerating={isGenerating}  onGenerationComplete={ ( ) => { } }  >  </ImageComponent> |

This is just the empty function.

1. Inside that inner function, check to see if the index has reached the amount of elements:

|  |
| --- |
| isGenerating={isGenerating}  onGenerationComplete={ ( ) => {  if(index === placeholderImages.length - 1) {  }  } }  > |

1. Only when this if statement is satisfied should we inform the rest of the program that the image generation process has stopped:

|  |
| --- |
| isGenerating={isGenerating}  onGenerationComplete={ ( ) => {  if(index === placeholderImages.length - 1) {  onGenerationComplete();  }  } }  > |

Notice that we have to call the function, not just pass a reference like before. It is now safe to do this since we know for sure that we are at the end of the array.

Appendix A – Creating a React app with Vite

First navigate to some folder you wish to work in. the first command will create a folder with the name you supply, here I am asking for a bc-final folder and app to be created.

1. npm create vite@latest **bc-final**  
   (it may ask you to install packages, just hit “y”)
2. Choose React for the next question
3. Choose JavaScript for the final question in this part

You should now have some instructions in the terminal window. In my case I need to CD into the bc-final folder and run two commands

After you run the commands the instructions on the terminal ask, you can open your code editor and point to this folder and the files inside.

Look in the package.json file for the command to run the actual application. In my case it is port 5173 yours may be different.

Appendix B – Definitions

**encodeURIComponent** is a JS function that is embedded in all JS versions. It is a security feature that ensures that only valid URL characters reach the server for processing. You can read about it here: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent>

A **BLOB** (binary large object) can be a varying-length binary string that can not exceed 2,147,483,647 characters long. It can also be binary data such as that used for composing images. We can also create a URL using the blob object. It is specially useful for uploading and downloading images.

The createObjectURL() static method is used with the BLOB type. In this case we have an image that needs to be displayed, so this is the only way to get that image to be shown on the browser.