

AI Image Generator Documentation

Project Name: AI Image Generator

Product Version: V1

Table of Contents

- Overview
- Front End
- Back End

Overview & Objective of the Project

OpenAI/ChatGPT is so trending now! Everyone is using it, and it seems like it is very powerful. I've been using it for some time, from understanding a specific concept with examples to generating code with proper prompts, and the result to me is very promising. So I was thinking, can it do more? Like generating images?

The purpose of building this website is to dive deeper into open AI APIs such as DALL-E and build a full-stack website of an image-generating platform, where users can easily generate images and choose to share them with the community & download them.

As for the tech stack, I'm planning to use Node.js, Express, and MongoDB to establish my backend server and database, and React.js + Tailwind CSS for the front end to complete this project as a MERN stack. I'm also going to deploy the website so that more users can enjoy their creativity!

The below content serves as a journal, to fulfill the purpose of:

- Keep track of the ordering of the project (which part to implement first, which to implement later), so that the reader has a basic understanding of how to complete a

MERN stack project from scratch.

- Keep records of detailed problems and how I solved them, so when I look back after months later, I can understand the logic behind why I used certain solutions to solve the problem.

Build Front-end Client Side

Set up

- Instead of “create-react-app”, after doing some research, I decided to use Vite, which is going to have some advantages below:
 - faster and leaner development experience for modern web projects
 - Vite has 2 major parts:
 - A dev server that provides rich feature enhancements over native ES modules
 - A build command that bundles your code with Rollup, pre-configured to output highly optimized static assets for production
 - the command to start the project is “`npm create vite@latest`”
- Setting up tailwind CSS - for the styling of our website
- Additional packages used in package.json: `file-saver`, `react-router-dom`

Start to create sample page components

- So far, we just need a home page and a CreatePost page.
 - tip: if you have a lot of components and you want to import them in a single file, you can do below:

```
create an index.js file within the page/components folder.  
import Page/Component-name from 'folder path'  
export { Page/Component-name }
```

Then you can include this line of code in the page that you want to render those components: `import { Home, CreatePost } from './pages'.`

- The header is composed of: a logo and a custom link for the use of “creating post”
- Basic route structure

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/create-post" element={<CreatePost />} />
</Routes>
```

- Components of the Home Page:
 - Card component to render image
 - FormField to allow inputs
 - Custom Loader component which is just an SVG file to render loading animations
- States in the Home Page:
 - loading: to check if the page is loading
 - allPost: all the posts on the home page
 - searchText: the search text in the form
- After finishing implementing all the logic in the front end and back end to create a post after ai generated the image, you can start with the logic to display those posts on the home page:
 - We have a `<Card />` component on the home page, so we need to implement this one in order to show the image on the home page.
 - What do we expect from the `<Card />` component:
 - When hovering over the card (CSS), I can see the prompt and who uploaded it, and also the option to download (a separate function) it.
 - Implementation of the `<Card />` component
 - First, we complete the “download” logic, we installed a “`file-saver`” npm package, and we can use the `saveAs` method, to download the image, the

syntax (from the official doc) is below:

```
FileSaver.saveAs("https://httpbin.org/image", "image.jpg");
```

- Then we will go back to our `<Card />` component:
 - Props to be passed in: since on the Home page, we use the map method and spread the entire post as a prop to the `<Card />` component, we can use all properties in the post such as `_id, name, photo, prompt`.
 - Elements in the component: image, prompt, name of the person who uploads the photo, and a download button.
- The last functionality is to implement a search bar, where you can search for images that have certain prompts.
 - The basic idea is to have an `onChange` event where every time you type in something, the front end is going to filter out the content and only display the images where those include either the name of the uploader or word(s) in the prompts.
 - Since you do not want the change event to fire every time you type in something, you can use `debounce`, to set up a timeout function, and only fire the filter step after 500ms of typing.

CreatePost Page

Part A. Finish the UI look on the “CreatePost” page:

- Since the create post page will have a functionality of randomly generating some images, we need to have a function to do that, we can create a new “utils” folder to store all utility functions.
- The create post page is going to have 2 forms/form fields sections:
 - A regular form where you can type in anything you want to generate
 - A random prompt form where it's just going to randomly fetch pre-seeded dummy prompts to generate an image

- We will use a custom `<FormField />` component to render it so that we do not repeat making HTML form fields.
- These form fields will be based on some conditions:
 - If you want to type in stuff by yourself, the regular form field will render
 - If you just want to randomly generate with pre-seeded prompts, the random button will generate
 - We can use an “`isSupriseMe`” variable to determine which form fields we want to render
- Below the two `<FormField />` , we are going to create our container for the image that's about to be generated.
 - Just to make the UI a bit better, we imported a “`preview`” image from our `asset` folder, and set up a condition: if we have a form photo, then the real photo will be rendered, otherwise, the `preview` image will be rendered.
 - To also perfect the UI, we will place the `<Loader />` component on top of the `preview` image, with the use of `generatingImg` , if we set it to true the loading effect will display.
- We are going to have 2 buttons with different usage:
 - First `button` is to generate the image
 - Second `button` is to submit our image so that we can share it in some sort of central hub

Part B. Finish the logics on the page (to be continued)

- `handleChange`
 - Use the spread operator `...` to spread the form properties
 - Set the `name` property to whatever you typed in the input
- `handleSupriseMe`
 - We implemented a utility function `getRandomPrompt` a while ago, the function is going to randomly fetch a prompt from our dummy prompt list, and we can now put that to use.

- Basically, the function is just going to change the `prompt` property in our `form` state, to a randomly generated prompt.
- `generateImage`
 - We are going to call our backend, and get the returned generated image, but so far we haven't developed our backend, so this would be a very good time to stop developing our front end and go developing our backend.
 - (After completing openai image generating post route, come back) - complete the data fetching logic with the built-in fetch API (or you can use Axios if you want).
 - A good practice is to isolate the URL that you are going to fetch, since after you deployed your backend, the URL will change, so if you have multiple fetch requests using this URL, you don't need to change all of them.
 - The form state has a photo property, how to set up this property so that the image can be rendered in the container? Check the below code:

```
const data = await response.json();

setForm({
  ...form,
  photo: `data:image/jpeg;base64,${data.photo}`
})
```

- Since no matter if we have an error or a successfully generated image, we want to set the `generatingImg` back to false, we can add a `finally` clause after the catch.
- `handleSubmit`
 - Now that we have finished the logics to interact with the openai API, it's time to think about how can we upload/submit our newly generated image to the cloud (Cloudinary), as well as to our backend database (MongoDB).
 - First, let's go to the backend, and implement our postRoute router.
 - Basic logic:
 - Step1: prevent the default behavior of the submit event

- Step 2: check if there're prompt and photo-generated, then set the loading state to be true, and fire the post action through the fetch API
- Step 3: After a successful post, then navigate to the home page
- Step 4: Remember to finish with error checking and a `finally` clause to set the loading state back to false
- After you implemented the logic for `handleSubmit`, you can go to the `Home` page, where you can write logic to fetch all the posts and render them on the home page.

Added Feature:

- Mode toggle
 - Setup: in tailwind config file, add `darkMode: 'class'`, also in the root `index.html` file, give the `body` element a `className` of "dark", so the system will know to use classes to enable dark mode.
 - created a button to toggle mode between "light" and "dark"

Building Backend Server Side

Set up

- `npm init` to bootstrap our server
- packages needed: `cloudinary, cors, dotenv, express, mongoose, nodemon, openai`
- to better refresh our server, we can change the script to:
 - `"start": "nodemon index"` - so that we are using `nodemon` to start our server and can automatically refresh the server whenever we have a change.
- to use the `ES6` import modules, we need to add a line of code in our `package.json` file - `"type": "module"` - so that we can use "import" and "export" just like we did in our front end.
- create an index page in server folder, and import necessary packages, and initialize `express` app.

Database connection

- To structure our architecture better, we are going to create a separate folder to store all database related files
- In the db file, we just simply create our connection logic with `mongoose`, and export it so that we can use it in our main index file.
- In our main index file, before the app can start listening to a port, we need to connect to our database.
- The mongoDB URL is going to be stored in the `.env` file - this is a security consideration since when we push our code to github, the `.env` file will not upload.
- The last step is to create our `Post` model, pretty standard, just create the Schema and the model and export the model. The model will basically be the “bridge” to interact with db.

Creating routes for our backend

- The routes folder should be separate from the db folder, it is a best practice we separate our folder structure by isolating functionality.
- There're two types of routes:
 - Post routes where you can submit your ai-generated image to the community (store it in the db)
 - Dalle routes, which is going to call the dalle API
- Post route:
 - We are going to use the `cloudinary` Node SDK - The Cloudinary Node SDK allows you to quickly and easily integrate your application with Cloudinary. Effortlessly optimize, transform, upload and manage your cloud's assets.
 - We are also going to use the Post model here since we are going to upload our image to Cloudinary.
 - Have your Cloudinary API name, key and secret stored in the `.env` file as environment variables, and you can use them to config your Cloudinary like below:

```
cloudinary.config({
  cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
```




```
api_key: process.env.CLOUDINARY_API_KEY,  
api_secret: process.env.CLOUDINARY_API_SECRET,  
})
```

- We need 2 types of routes:
 - GET all posts:
 - Just use `Post.find({})` - will fetch all posts from db
 - CREATE a post:
 - Step1: get all the properties (name, prompt, photo) from the request body
 - Step 2: use `cloudinary.uploader.upload(photo)` to get the cloudinary image url
 - Step 3: create a new Post instance with the URL you just got
 - Step 4: send back the newly created post through a response
 - Don't forget to wrap the logic in a `try-catch` block to improve fault tolerance.
 - We may need more such as deletion, but it only makes sense if we implement authentication & authorization, which could be a future project.
 - After the implementation of the backend routes is finished, we can then go back to our front end - CreatePost page, to implement one last step: our `handleSubmit` function to upload and share the photo.
- Dalle route:
 - We are going to use the `openai` package here - The OpenAI Node.js library provides convenient access to the OpenAI API from Node.js applications, since we will utilize the `openai` api to generate our image.
 - To have the `openai` generating image for you, below is the format, or you can check the docs here:

OpenAI API

An API for accessing new AI models developed by OpenAI

 <https://beta.openai.com/docs/api-reference/images/create>

```
const aiResponse = await openai.createImage({
  prompt,
  n: 1,
  size: '1024x1024',
  response_format: 'b64_json',
})
```

- The more detailed the description, the more likely you are to get the result that you want
- Breakdown of the properties:
 - prompt - the prompt you input in your front end
 - n - the number of images to generate, default to 1
 - size - the size of the generated images, must be one of:
`256x256` , `512x512` , or `1024x1024`
 - response_format - the format in which the generated images are returned, either `url` or `b64_json`
- To convert the ai response to an actual image, we need to run this line of code (from the docs:):
 - `const image = aiResponse.data.data[0].b64_json`
 - And we can then send the image back through the response json object
- After you have completed this post route, you will be able to get the image back to your front end, so this is a perfect time to go to the front end and test it.