Group Root 9

Paletä Project Report

Github Repo

TABLE OF CONTENTS

Contents

PROJECT OVERVIEW

For our project we chose to create Paleta, a tool used for obtaining colour palettes from a variety of methods. Paleta is designed to cater to visual artists and individuals seeking creative inspiration. Whether you need colour palettes for work or personal reasons, we are certain Paleta will be able to help.

Milestone 1:

Initially we had planned to have 2 main features. One of which being the colour palette generation from uploaded images, and the other being an option to create a moodboard with the colors that you extract from an image. Eventually while working through the project we decided to drop the moodboard feature, and instead have our second main feature be a colour palette generator driven by AI generation.

Key Feature 1:

Our first feature is Image-Based Colour Extraction, where we use ColorAPI in order to extract the most dominant colours from user-uploaded images and identify those colors in a variety of formats. This feature was created with React, as it allowed our group encapsulate components of the web application with ease, and making changes to affect the entire web page went smoothly.

Within the colour extractor, we have these features:

- Option to drag or upload an image saved on the users PC
- Select the number of wanted colors both before and after uploading an image, from the options 4, 6, 8 ,10
- Colors shown in order of prevalence in the image.
- Colors have their name, hex, RGB, and cmyk values shown, as well as a square of the color being displayed on the screen.

Key Feature 2:

Our second feature is Prompt-Driven Palette Generation, where we use OpenAI in order to generate a color palette based on the prompt which the user gives.

Within the prompt-driven palette generator, we have these features:

- Ability to see chat history with AI, as well as the responses that the AI gives directly.
- Select the number of wanted colors both before and after entering a prompt, from the options 4, 6, 8, 10.
- Colors are shown in order of prevalence in the image.
- Colors have their name, hex, RGB, and cmyk values shown, as well as a square of the color being displayed on the screen.

SDLC MODEL

The SDLC we decided to use throughout the project was Agile, and in specific the Kanban framework of Agile.

The decision to opt for the Kanban model under the Agile methodology was influenced by its visual nature, which allows for an immediate understanding of the project's status, work in progress, and what tasks are upcoming. By using a Kanban board, team members can easily visualize the flow of tasks, pinpoint bottlenecks, and allocate resources more effectively.

We used Notion to represent the Kanban board digitally, and to represent our tasks. In essence, the combination of the Kanban model with Notion serves a dual purpose. First, it aligns with our goal of maintaining transparency and agility in our development process. Second, it provides an efficient tool that caters to the team's needs, ensuring that tasks are monitored, managed, and executed in a streamlined manner.

After using Kanban for most of the project lifespan, it is safe to say that there were definitely things that could have been improved. Starting with what went well, having the Agile methodology allowed us to be flexible in our project plans and API choices. We ended up switching up many of the APIs and general structure of the project as we were working on it which may not have been possible if we were using a more formal structured methodology. We also had flexibility to adapt our meeting schedules to each persons classes and things that they had to do. Kanban also allowed for clear visual task planning whenever we had tasks to put on the board.

On the other hand, there were aspects of the SDLC which did not work as well as we had expected, and could be used as opportunities to learn from them. Our meetings were fairly consistent, but could have been even further extended to perhaps daily/bi-daily meetings in order to keep information more up to date with the team. Although using the Kanban framework, our group didn’t exactly keep up to date with putting tasks on the Kanban board, and rather just spoke to each other in our communication channels. By having a more proper platform created for group collaboration it’s possible we could have improved the number of tasks we ended up putting on the notion board. This also extends generally to our github page, which also did not receive many issues on it, and most issues were communicated through Discord.

APIS & High Level Features

01

ColorAPI

- Comprehensive Color Data
- Interactive Backgrounds

02

OpenAI API

- Creative Content Generation
- Pattern Recognition

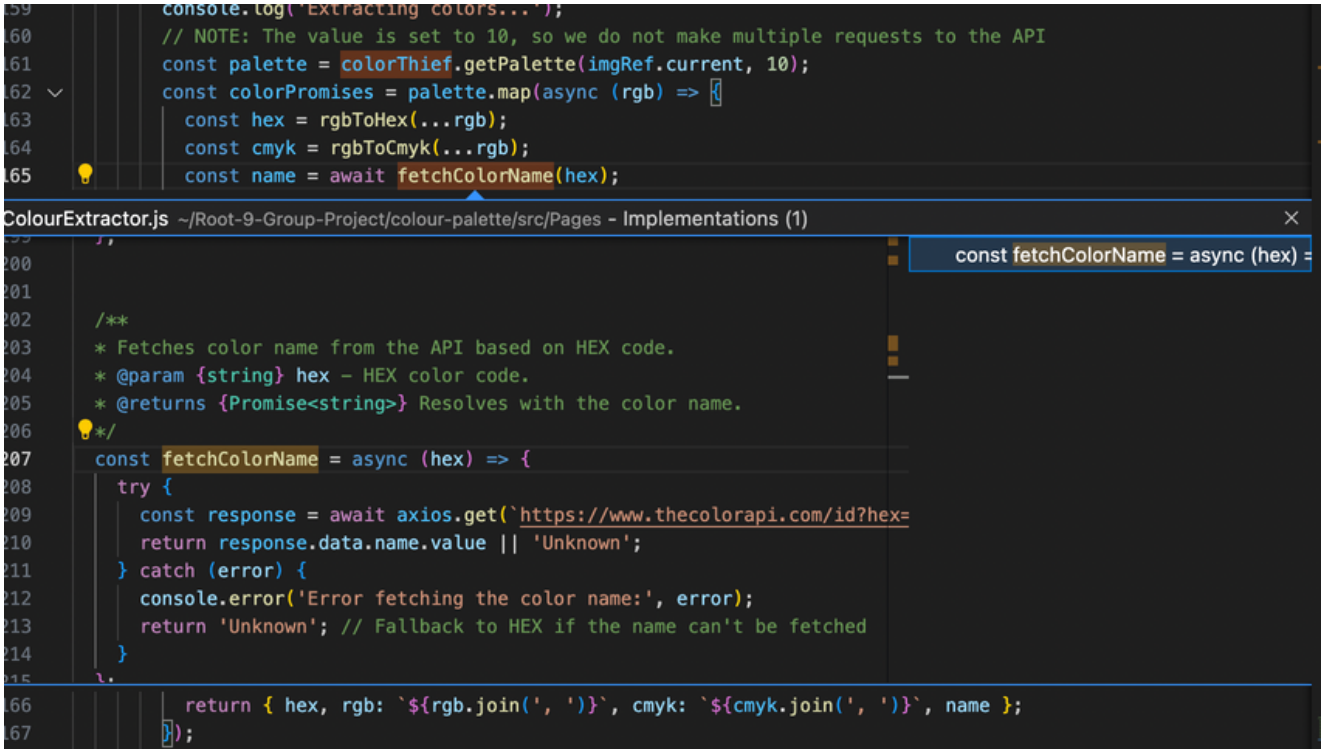
OpenAI

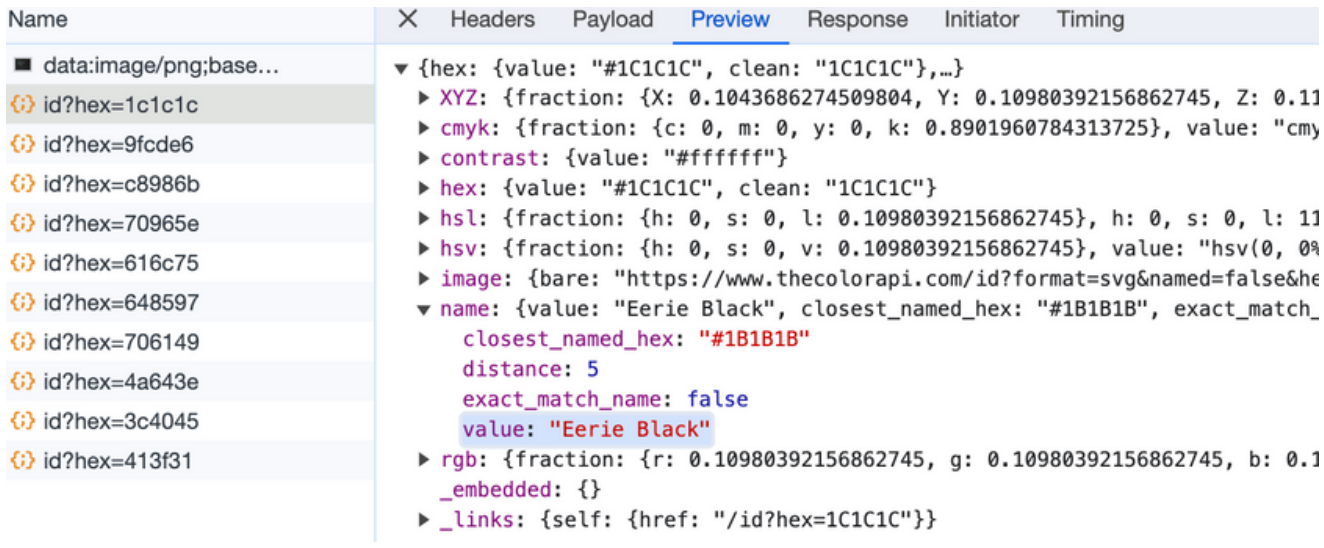
node JS

COLORAPI

This API is the core of the colour extraction that we do with the image upload feature. It provides both the functionality in terms of identifying the colours as well as giving the colours in a variety of formats.

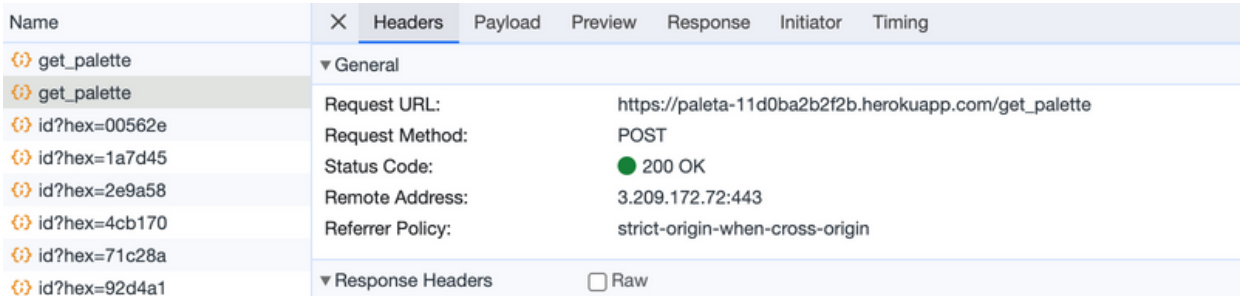
First we have used the react library: ColorThief
to get 10 hex values in one call based on a picture input. For each value, we call the function fetchColorName to fetch the colour name using Colour API. Internally, we have written two functions to convert RGB into HEX and CYMK.

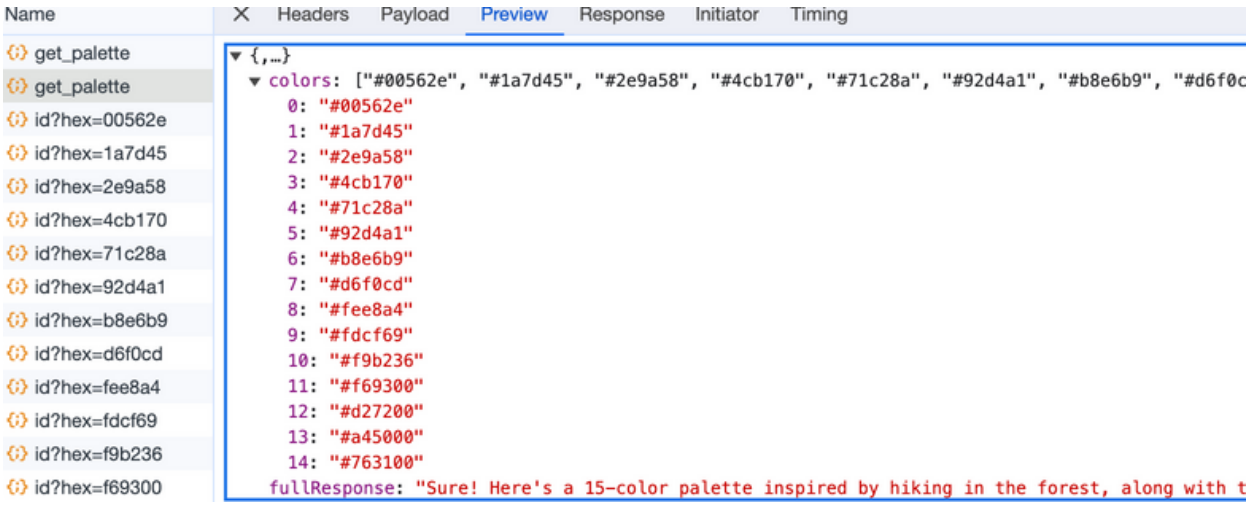




OPENAI API-CLIENT SIDE

In order to call the OpenAI API, which requires authentication, we use a back-end call through a Heroku App that we developed in order to make the request secure. The response will return 15 hex values which we then decode into RGB and CYMK using our own calculations. Also, the colour API, as described above, will be invoked for each hex value to retrieve the colour name.






```
39    try {
40      const apiUrl = 'https://paleta-11d0ba2b2f2b.herokuapp.com' || 'http://localhost:3000';
41      const response = await axios.post(`${apiUrl}/get_palette`, { prompt });
42
43      const fullResponse = response.data.fullResponse;
44
45      // Extracting colors and converting them to the desired format
46      const colorPromises = response.data.colors.map(async (hex) => {
47        const rgb = hexToRgb(hex);
48        const cmyk = rgbToCmyk(rgb.r, rgb.g, rgb.b);
49        const name = await fetchColorName(hex);
```

PaletteGenerator.js ~/Root-9-Group-Project/colour-palette/src/Pages - Implementations (1)

const fetchColorName = async (hex) =>

```
232   };
233
234   /**
235   * Fetches color name from the API based on HEX code.
236   * @param {string} hex - HEX color code.
237   * @returns {Promise<string>} Resolves with the color name.
238   */
239   const fetchColorName = async (hex) => {
240     try {
241       const response = await axios.get(`https://www.thecolorapi.com/id?hex=${hex.replace('#', '
242       return response.data.name.value;
243     } catch (error) {
244       console.error('Error fetching the color name:', error);
245       return hex; // Fallback to HEX if the name can't be fetched
246     }
247   };
```

```
50        return { hex, rgb: `${rgb.r}, ${rgb.g}, ${rgb.b}`, cmyk: `${cmyk.c}, ${cmyk.m}, ${cmyk.y}, ${cmyk.k}`, name };
51      });
```

OPENAI API-SERVER SIDE

We added a CORS header to allow our app hosted on netlify to call OpenAI through the Heroku NodeJS back-end.

```
// Enable CORS for requests from the specific origin (update with your React app's domain)
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "https://mypaleta.netlify.app"); // Allow all origins
  res.header(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept, Authorization"
  );
  if (req.method === "OPTIONS") {
    res.header(
      "Access-Control-Allow-Methods",
      "POST, PUT, PATCH, GET, DELETE"
    );
    return res.status(200).json({});
  }
  next();
});
```

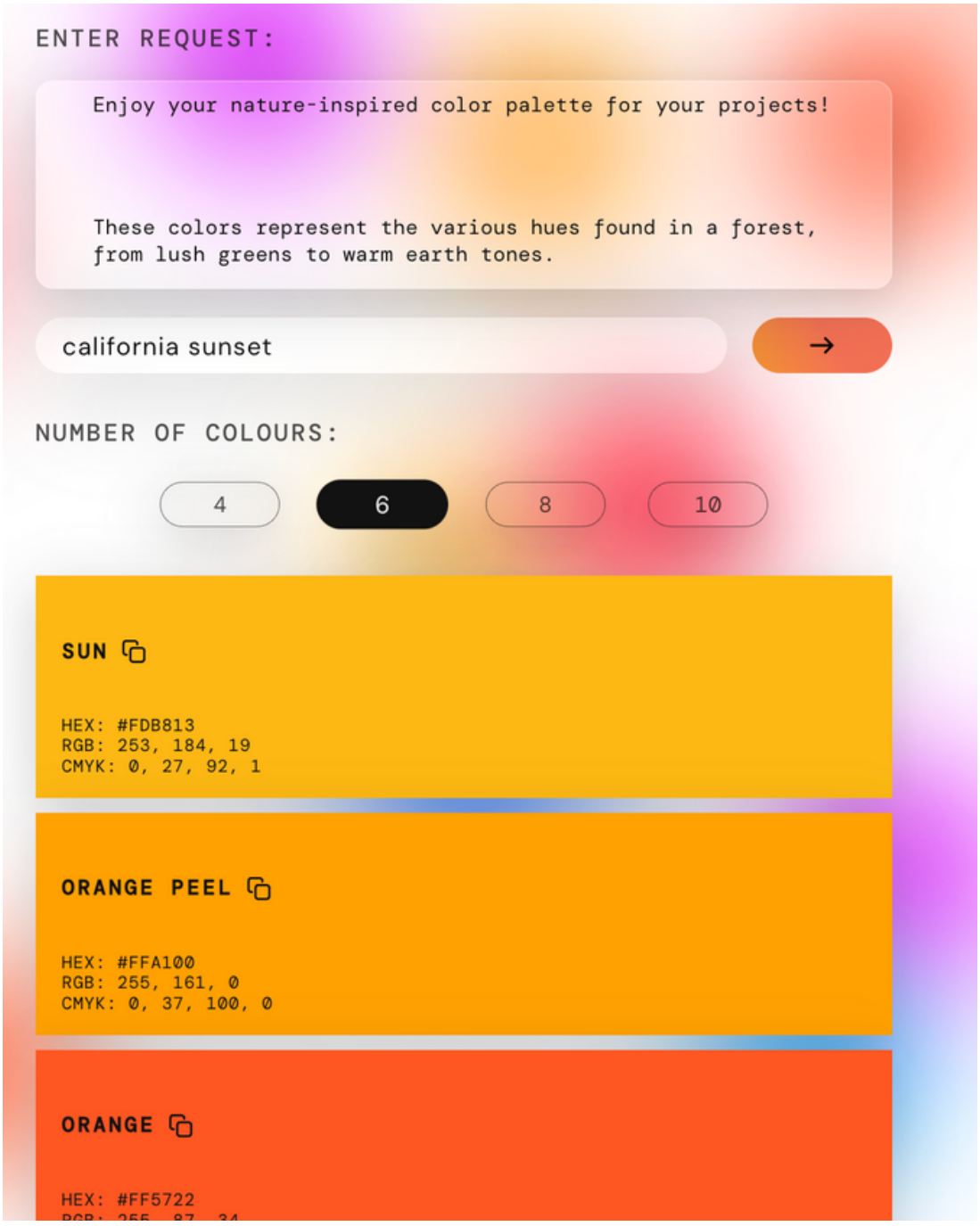
We combined the user's input with a prompt "Provide a 15 colour palette with HEX codes no colour name needed", in order to get the response we expected.

```
app.post('/get_palette', async (req, res) => {
  const userPrompt = req.body.prompt + " Provide a 15 color palette with HEX codes no color name needed.";
  const maxTokens = 300; // Maximum number of tokens for the response

  try {
    const response = await axios.post('https://api.openai.com/v1/chat/completions', {
      model: "gpt-3.5-turbo",
      messages: [
        { "role": "system", "content": "You are a helpful assistant." },
        { "role": "user", "content": userPrompt }
      ],
      max_tokens: maxTokens
    }, {
      headers: {
        'Authorization': `Bearer ${OPENAI_API_KEY}`
      }
    });
```

OPENAI API-SERVER SIDE

When we get user's response from prompt, we will get a long response with hex numbers. So we use a variable called HEX_COLOR_PATTERN to filter all the hex numbers by calling match function, which functionally to remove the hex numbers to maintain a clear message in order to make our AI tool more lively and show a clean and neat final message to our user window.



Code Tests

Our project has two sets of tests, unit and integration tests. They will be outlined in the two overhead sections below.

Unit Tests:

Our unit tests confirm that independent functions are working as designed using Jest and React testing libraries.

Toast - ensure modal dialog boxes are displaying the expected messages

Number button - check that event is fired when a number selection is made

colorutils - confirm success / failures of translations for colour codes. Ie: hex <-> rgb

Integration Tests:

Our integration tests confirm that user actions on the web page flow to the components and work together as expected.

Fetchcolor - mock the HTTP GET method of the ColourAPI to ensure response schema is in the expected format

Navigationbar - ensure routing is going to the correct pages back on navigation bar clicks

Background color - ensures browser local storage cache is set and cleared appropriately

CI/CD Infrastructure

Our CI/CD Infrastructure is headlined by using Github Actions for continuous integration and Netlify / Heroku for the continuous deployment. We set up the Github Actions by including a workflow folder and having a yml file which installs needed dependencies and runs the test files using the command npm test. Once the tests are complete, deployment is done in two steps. The first step is to deploy the static parts of the web application with Netlify, and then we deploy the server parts with Heroku. Taking the Heroku deployment and entering it as an environment variable, we can connect both parts in one website, making users have free access to all features in one place.

01

Run Workflow File

Workflow file runs the test command and the program is built and the tests are ran.

02

Send Approved Static Build to Netlify

The static build is sent to Netlify and deployed. This only accounts for the static parts of the website, and the rest are deployed elsewhere.

03

Send Approved Server Build to Heroku

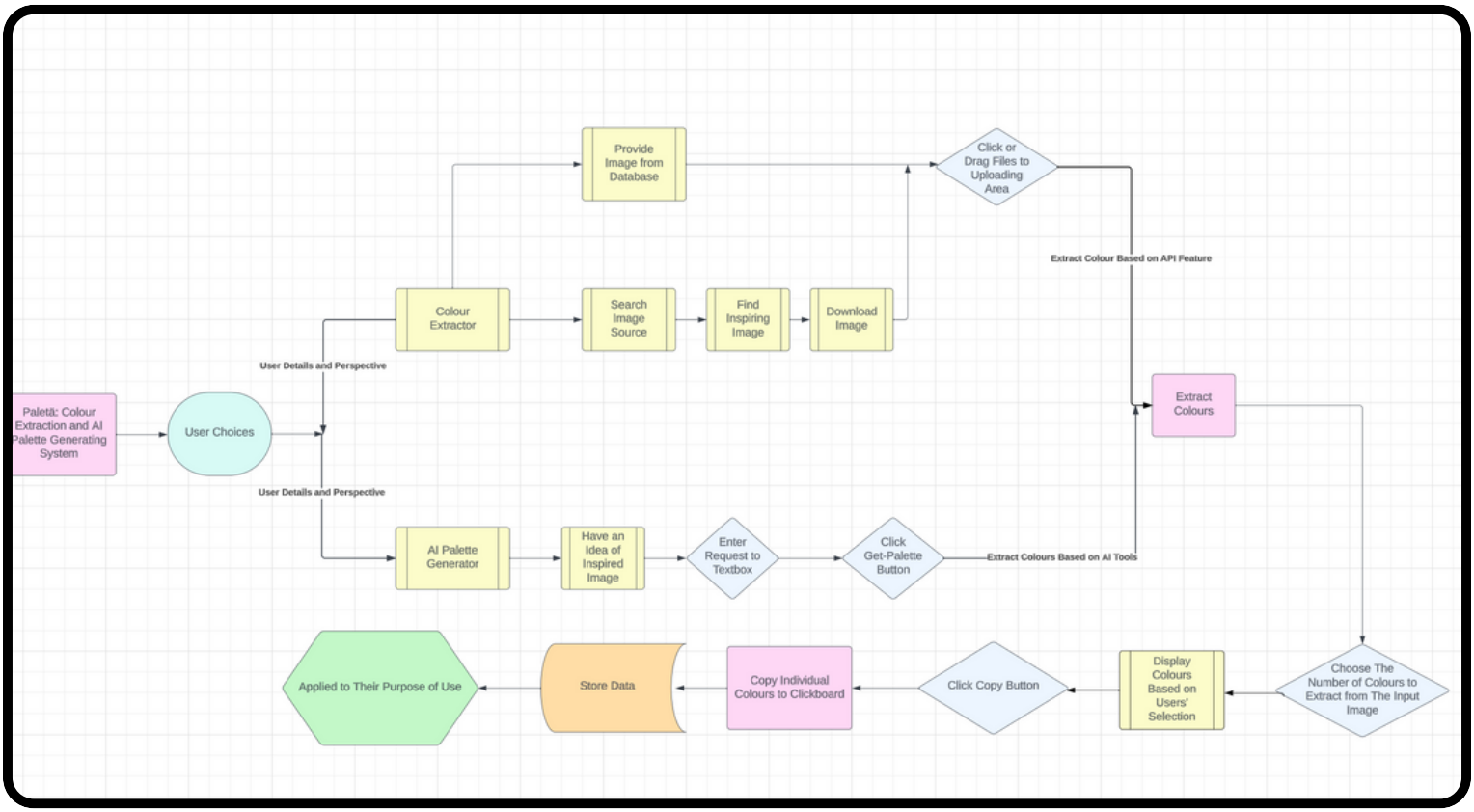
We send the back-end server build to Heroku and it deploys the features which are associated with the server side.

04

Website is fully deployed.

Connecting the Heroku deployment to Netlify using environment variables, the website is fully deployed and accesible through the Netlify link.

High level data flow diagram



Project Takeaways

This project was styled in a way that is fairly unfamiliar for most students doing this course. For our group this was the first time doing a group project which required such close collaboration with one another.

Communicating	Throughout the course of our project, we encountered and addressed several notable takeaways and challenges. Group communication would be a significant challenge we faced, as transitioning from individual work to collaborative efforts seemed challenging at the beginning. However, after recognizing the necessity for enhanced communication within the team, we established regular timetables and meetings regularly to facilitate better coordination. .
Code Sharing	Simultaneous coding posed another challenge initially, leading to conflicts as team members coded simultaneously without proper communication. To address this, we implemented a more efficient workflow by creating branches for individual testing and utilizing pull requests, mitigating conflicts, and enhancing collaboration.
Test Cases	Due to limited initial knowledge of creating test cases, especially for UI elements, we spent a significant amount of time doing research to figure out how jest integrates with Github Actions. Eventually, we successfully connected the two tools and built a pipeline to ensure robust test running.
API Choices	In the process of selecting APIs, we initially opted for Cloudinary and Imagga Color Extraction based on early research. However, as we progressed with the implementation, it became apparent that these choices might not fully meet our project's requirements. Consequently, we pivoted to ColorAPI and OpenAI, aligning more closely with our project goals and delivering improved functionality.

Acknowledgements

Thank you for reading this report! Below is our group members, who worked on what, and other links.

- Project Overview: Anna
- SDLC Model: Taiga
- Features and APIs: Cindy
- Code Tests: Anna/Taiga
- CI/CD: Stefan
- Data Flow Diagram: Cindy
- Project Takeaways: Cindy
- Web Application Color Extractor: Anna/Taiga
- Web Application AI Palette Generator: Anna/Taiga
- Web Application UI: Anna
- Project Manager: Taiga
- Presentation: Team
- Report: Team

CONTACT

Stefan Pricope (scp10@sfu.ca)
Taiga Okuma (toa12@sfu.ca)
Cindy Xiao (yxa143@sfu.ca)
Anna Rusinova (ara103@sfu.ca)
GitHub Repository: Linked Here
GitHub Server Repository: Linked Here