**# Colors Challenge (fullstack)  
  
Steps to run code:**

git clone https://github.com/Ajinkya710/colorSwatch.git

cd colorSwatch

**Install dependencies:**

**Frontend**

cd frontend

npm install

npm start  
This will start the frontend on <http://localhost:3000>.

**Backend**cd backend

pip install -r requirements.txt  
python manage.py test

python manage.py runserver  
This will start the Django server on <http://localhost:8000>.

API - <http://localhost:8000/api/colors>

**Stage 1:**

**Frontend flow:**

**The frontend service for supporting RGB and HSL color swatches is as follows:**

**src/Pages/ColorSwatch:** This directory contains the main page displaying the color swatches.

**Index.tsx:** The ColorSwatch component that displays five color swatches using RGB and HSL values.

**src/Pages/ColorSwatch/store:** The Redux store specifically for the ColorSwatch page components.

* **action:** Defines actions for API calls to the backend to fetch color-related data.
* **selector:** Contains selectors to retrieve state slices related to the color swatches from the Redux store.
* **slice:** A slice of Redux state that holds data specific to the ColorSwatch page, such as the fetched color data.
* **types:** Type definitions used throughout the ColorSwatch page, including color schemes and data structure definitions.

**src/helper:** Contains helper functions responsible for normalizing and generating color codes (e.g., converting RGB to HSL or vice versa).  
  
**Backend flow:**  
**API Endpoint**

**GET /api/colors/** : Returns a list of randomly generated color swatches (RGB, HSL)

**utils.py:** Defines a decorator (register\_color\_generator) to register different color generation functions.

The color generation functions (generate\_rgb\_color, generate\_hsl\_color) return random color values.

The generate\_random\_color function generates a list of random colors using these registered generators.

**views.py:** The ColorSwatchView handles the GET request to /api/colors/swatches/ and responds with a list of generated colors.

**urls.py:** The /api/colors/ endpoint is exposed to the frontend for fetching random color swatches. **Stage 2**

**Now, in order to add support for new color spaces to our existing code, follow our api pattern and create a response type, in case of brgb - it would look like:** {

"type": "brgb",

"red": 255,

"green": 255,

"blue": 255

},

**Backend changes:  
Just register brgb color generator function that return brgb response type**

@register\_color\_generator('brgb')

def generate\_brgb\_color():

return {

'type': 'brgb',

'red': random.randint(0, 128),

'green': random.randint(0, 128),

'blue': random.randint(0, 128)

} **Frontend changes:  
Now, we need to add brgb support to generate brgb color at frontend. Just add brgb color string generator in:   
  
Step 1: src/Pages/ColorSwatch/store/types.ts**export const COLOR\_TYPES = ["rgb", "hsl", "brgb"]; //add brgb  
  
export const COLOR\_SCHEMES: Record<ColorType, {

properties: string[];

formatter: (color: any) => string

}> = {

// add brgb color generator function

brgb: {

properties: ["red", "green", "blue"],

formatter: (color: { red: number; green: number; blue: number }) =>

`rgb(${color.red}, ${color.green}, ${color.blue})`,

},

};

**Step 2: Add Normalizer function in helper.ts**export const normalizeColor = (color: { [key: string]: number }, type: string) => {

switch(type) {

…

case 'brgb':

// Normalize BRGB values to RGB (0-255)

return {

red: Math.round((color.red / 10000) \* 255),

green: Math.round((color.green / 10000) \* 255),

blue: Math.round((color.blue / 10000) \* 255),

};

…

}

};  
 **Summary**Ensure the “brgb” response follows the specified structure.

**Backend:** Register a brgb color generator using the decorator pattern.

**Frontend:** Implement a color formatter for brgb in types.ts and add to COLOR\_TYPES.  
 Add Normalizer function for the brgb color scheme in normalizeColor function.