Gbiv System Design Specification

Release 1.0

Dux D-zine

Nov 17, 2022

TABLE OF CONTENTS

1 System Overview **1**

2 Software Architecture **2**

3 Software Modules **5**

3.1 Frontend Manager 5

3.2 Web Interface 8

3.2.1 Main Page (Upload an Image) 10

3.2.2 Example Palettes Page 10

3.2.3 Color Theory Page 10

3.2.4 About Us Page 10

3.3 Backend Manager 10

3.4 Color Analysis 13

3.5 Palette Database 15

3.6 Database Interpretor 17

4 Dynamic Models of Operational Scenarios **20**

5 Acknowledgments **23**

CHAPTER

ONE

SYSTEM OVERVIEW

Gbiv is a web application with the goal of making color theory more accessible to the world and inspiring the designer in all of us. Its primary use case is that users can upload images and Gbiv will extract the dominant color and use it to suggest related colors and palettes. In addition to this, Gbiv shows users example palettes that have been viewed often on the site.

Like most web applications, our system is divided into frontend and backend. On the frontend we have a "frontend manager" module which employs the Flask framework and python code to build the general structure for our user-facing website. The site itself utilizes HTML, CSS, and Bootstrap for styling and extra functionality in terms of user interface. We use the common page system and employ 4 pages: a main page for users to upload images, a page to see example palettes, an informative page about color theory, and an about page describing the project.

On the backend we have a similar "manager" module which also uses Flask and ties together the front end framework with scripts and a database in the backend. The scripts that contain functions that are called by Flask are divided into two python files: the color analyzer and the database interpretor. The former handles all the color extraction and related color calculation while the latter provides an interface with the database that stores example palettes. The database is implemented using MongoDB and is accessed using Mongo's provided python library.

These two sections of the system are tied together using the Flask framework as described above. Gbiv is hosted online using Firebase and the source code is maintained through GitHub provided version control. For more information there are several documents describing Gbiv: the SRS, the SDS (this document), the user documentation, and the developer's manual.

CHAPTER

TWO

SOFTWARE ARCHITECTURE

The divide between frontend and backend was a guiding force in designing our software architecture. On one hand, it was important to maintain modularity within the two sides for ease of development. On the other, it was key that the two sides communicate fluidly so that the entire system could function properly. This led us to creating a "manager" module for both sides and use Flask as our framework throughout.

The diagram below shows our architecture visually. Note that there is a third section of the application shown which is dubbed the "User Space." This is not part of our implementation, but is an essential component of our design because ultimately Gbiv is built for the users.

The model above shows both components of the system and their interactions. We can further elaborate our architecture by focusing on each of these dimensions in particular. First, we can examine the individual components using the following table:

Table 2.1: Software Architecture Modules and Sub-Modules

Module/Sub-Module	**Category**	**Functionality**
Palette Database	Backend	Stores popular palettes that many users have viewed.
Database Interpretor	Backend	Interfaces with database to pass queries from the system to the DB and to transfer data from the DB to the rest of the application.
Color Analyzer	Backend	Extracts dominant color from images, finds related colors, generates relevant palettes.
Backend Manager	Backend	Communicates with frontend and sends control messages to backend modules.
Frontend Manager	Frontend	Communicates with backend and sends control messages to frontend modules.
About Us Page	Frontend	Tells users about the team and the project.
Main Page (Image Upload)	Frontend	Allows users to upload image files and view dominant color, related colors, and relevant palettes.
Color Theory Page	Frontend	Gives users more information on the basics of color theory so that they understand what Gbiv is providing.
Popular Palettes Page	Frontend	Displays popular palettes in the Gbiv database to give users ideas and inspiration for their design projects.
Web Server	Frontend / User Space	Bridges the gap between the application and the user space. Hosts Gbiv files and provides infrastructure for the application to be accessed via web browsers.
Internet	User Space	The network infrastructure that allows users to access the application.
User	User Space	Anyone and everyone who has an interest in design and/or color theory.

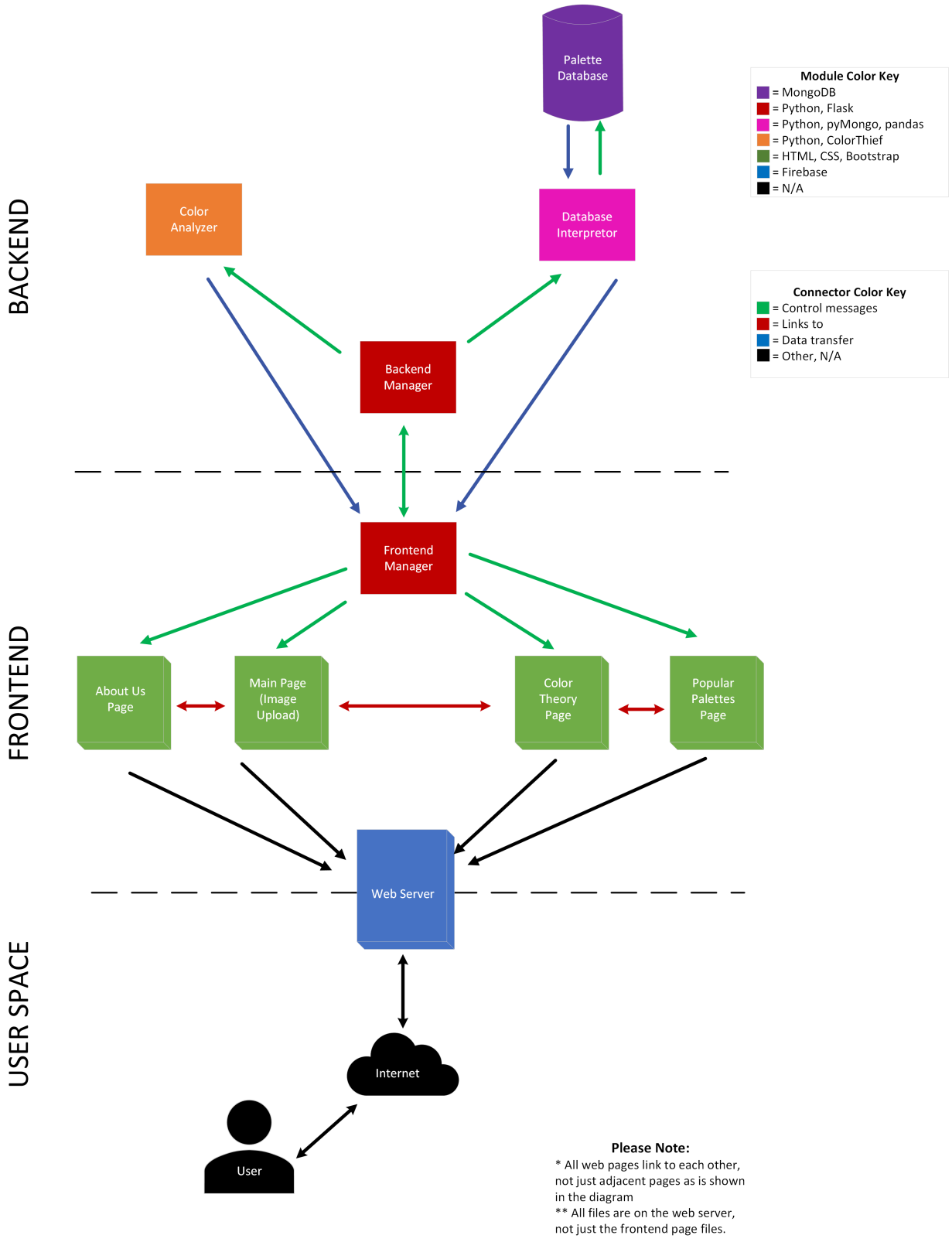

Fig. 2.1: Gbiv Software Architecture

Looking at each component in isolation is one way of viewing a system, but equally important to the application is the interactions between these components. The software architecture design shows all communication between modules, but we can highlight a few of the key interactions to better understand how Gbiv will function.

First, we can look at the transferring of control messages between the frontend and backend manager modules. These messages are formatted with the Flask python framework which is also used in implementing both of these key modules. Having consistent implementation and technology in these "bridge" modules allows communication without complicated translation or an additional framework in the mix.

Another key interaction we can profile is the web server's interface with both the front end and user space. Web hosting is what allows us to easily provide the functionality of Gbiv to end-users. We decided to use Firebase for our web hosting because it allowed us to outsource server side logic and networking while maintaining the unique design of our web page. Web hosting is a great way to reach users because it provides an interface that is easily accessible and familiar to the majority of the target users.

CHAPTER

THREE

SOFTWARE MODULES

Below are in depth descriptions of the design of each individual software module. It should be noted that not every component of the software architecture (see Fig. 2.1) is included in this list because not every component is part of the system we implemented. In particular, we do not describe users (for more information on users see the SRS), the Internet, or our web server (for which we used a OOTBS/COTS).

3.1 Frontend Manager

The purpose of the frontend manager module is to help bind together separate components of the system. In particular, it is one half of the duo that is responsible for bridging the front and the back end (for more information on the other component of that connection see Section 3.3). Because the frontend manager acts primarily as a middle ground for passing information and control messages, it does not have much internal structure. However, we can still represent this module in a static fashion using the diagram below. Note that there is still interfacing with other modules in this model because the frontend manager is defined by its interface with other parts of the system.

The frontend manager acts as a sort of "ambassador" between the frontend and the backend which means it accepts inputs and outputs from both sides. On the backend side, this module gives outputs to the backend manager and receives inputs from both the color analyzer and database interpretor modules. The outputs that are given to the backend manager are control messages that indicate what the user is "asking for," which results in the proper data being generated. The inputs from the backend are different types of data intended to be delivered to end-users. Because this module is implemented using the same framework and technologies as the backend, the data can be transferred directly from where it is generated to the frontend manager.

In terms of interfacing with the frontend, we can again divide the interactions into inputs and outputs. Inputs from the frontend originate from user interaction which is translated into function calls to the frontend manager. Outputs to the frontend carry essentially the same information that has been generated in the backend, but it is important that the data is formatted for wed display so that it can be utilized to change the website graphically. A dynamic diagram can be used to show these interfaces in a more streamlined way:

The design rationale behind the frontend manager can be summarized as: optimizing system communication. We could have designed Gbiv so that frontend components interfaced directly with backend modules, but by establishing a central place for inputs from and outputs to user-facing modules, we simplified our implementation significantly.

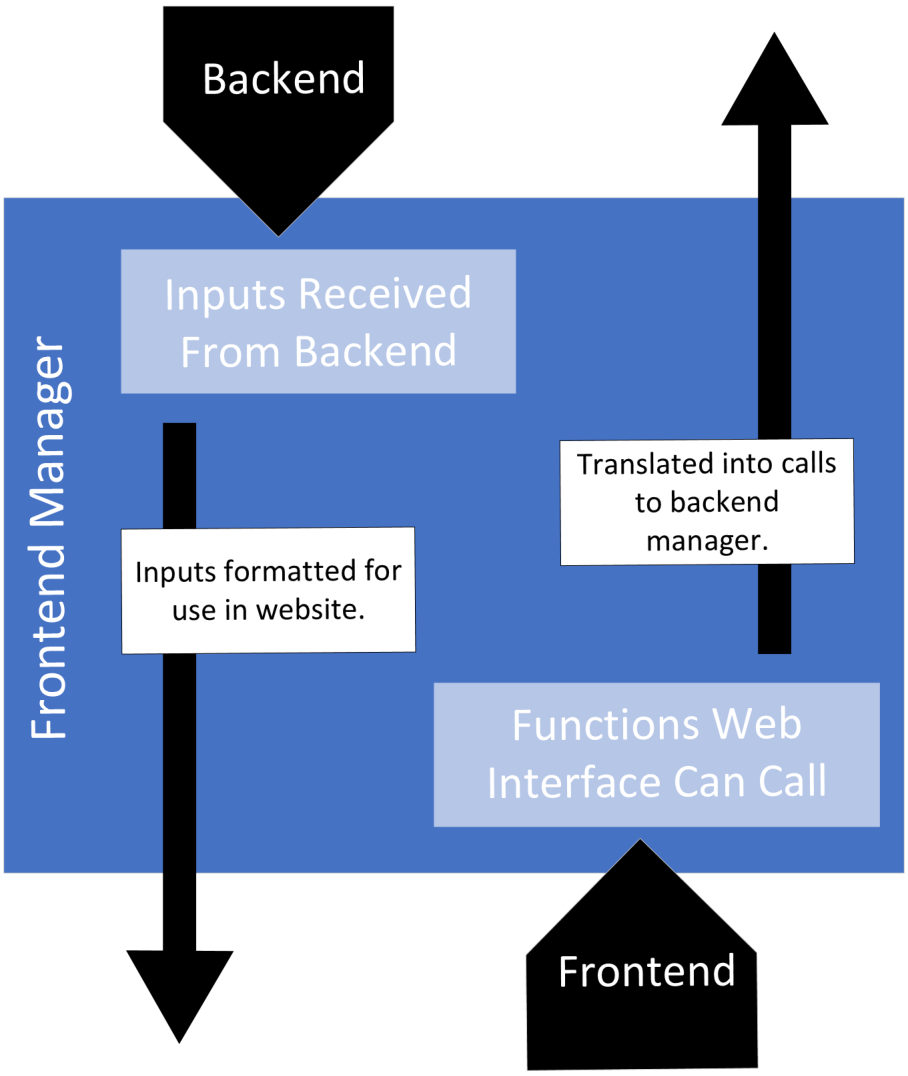

Fig. 3.1: Frontend Manager Static Model

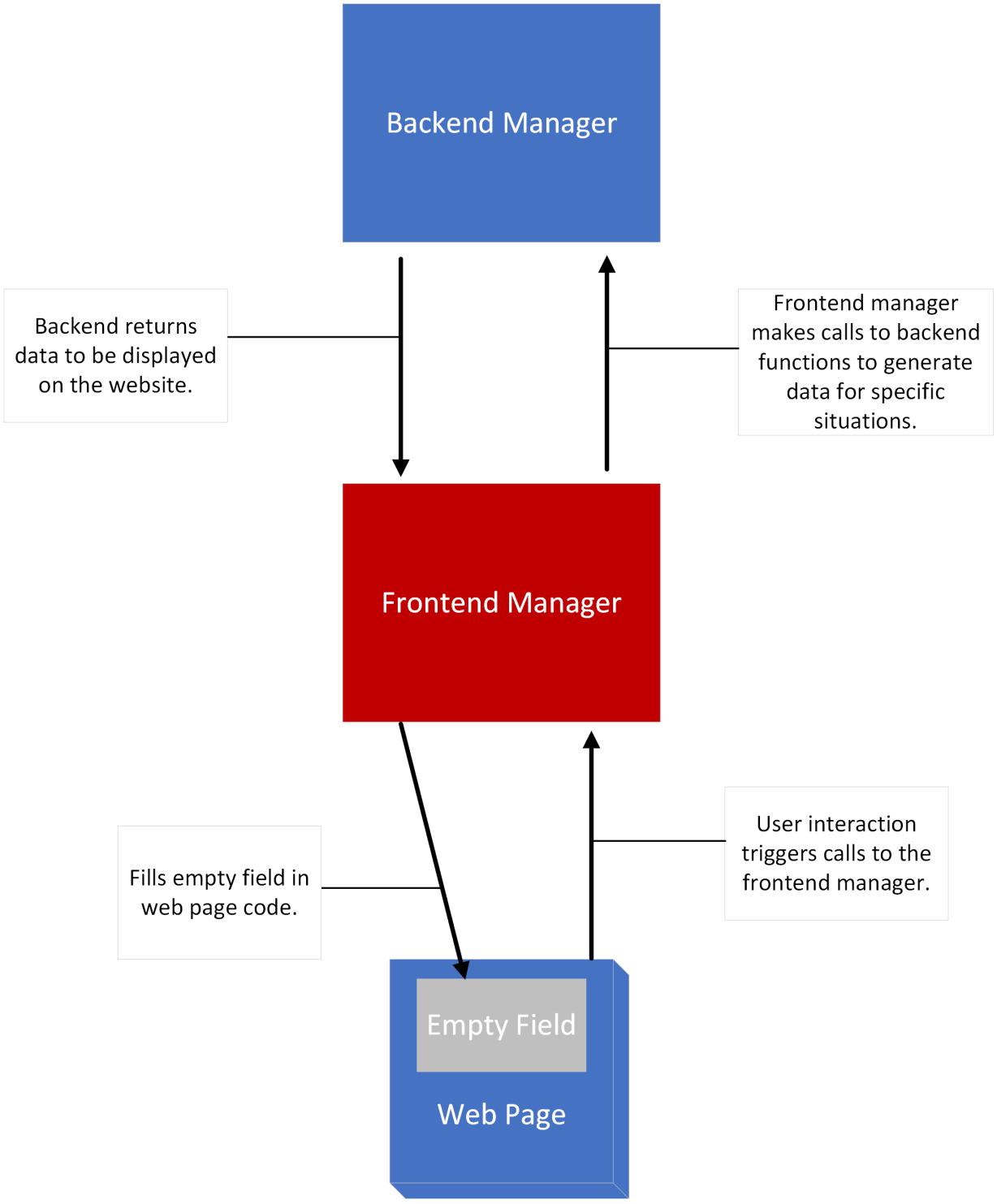

Fig. 3.2: Frontend Manager Dynamic Model

3.2 Web Interface

The Web Interface module is essential to the system because it defines the user experience in our application. The design of our website follows the traditional multi-page model seen on many popular websites with a navbar at the top to navigate between the pages. Each page has a different functionality and displays different information to the user. The general layout of each page is shown visually in the static diagram below:

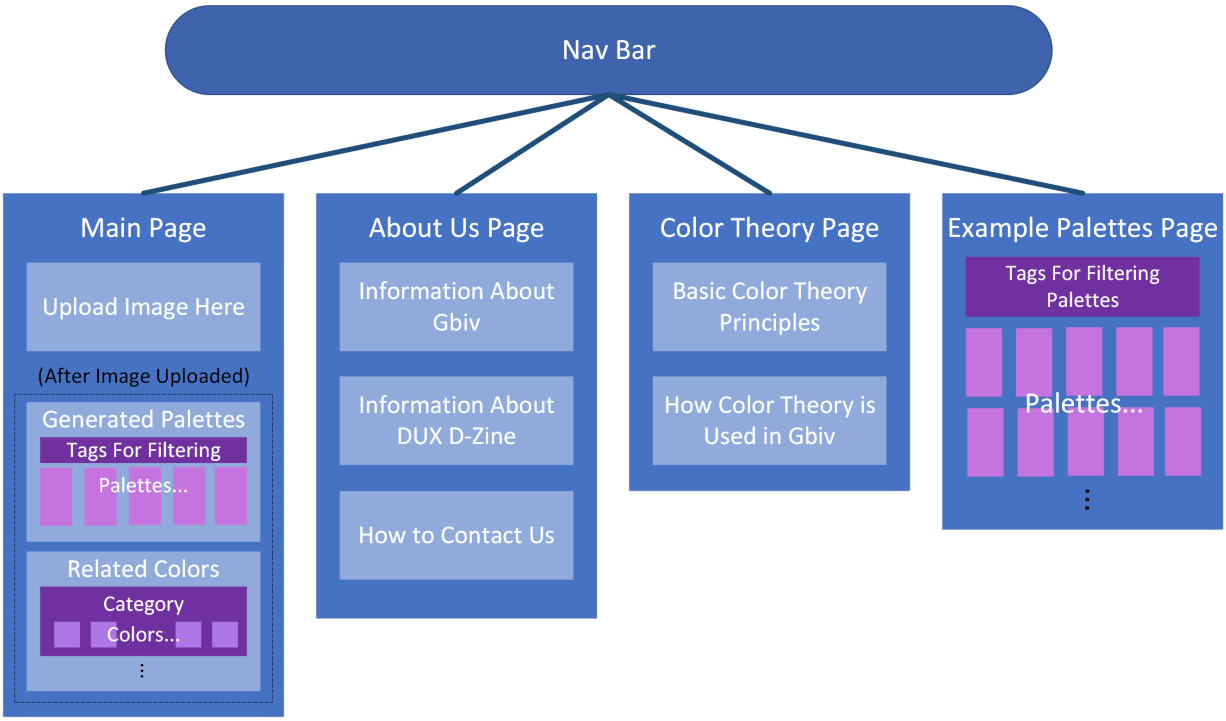

Fig. 3.3: Web Interface Static Model

There are two main interactions that the web interface has with other components in the system. First, it interacts extensively with the users through Internet by way of the server it is hosted on. The web interface is how the user requests the services that Gbiv provides and receives the information that is generated in response to these requests. Gbiv is a web-hosted application, which makes the server that hosts our application absolutely vital in the interface between users and the website module.

The second interaction that is key to keep the web interface functioning is the communication between each page and the Frontend Manager module. The Frontend Manager connects the user-facing display with the backend functionality and allows for dynamic pages that change in response user input. These two types of interaction are elaborated on in the dynamic model below:

The website was designed in this way for two primary reasons: (1) the multi-page website is familiar and therefore easily navigable for our users and (2) having separate pages increases modularity. The former point is important for Gbiv because our target users are a very wide demographic, so we want an intuitive and accessible user interface. The latter point has important advantages when it comes to the development of the system. In particular, modularity allows for easier delegation of tasks and for more efficient and focused debugging when problems arise.

The web interface module can be divided into sub-modules based on separate pages on the site. Below we have a brief description of each page's functionality and structure.

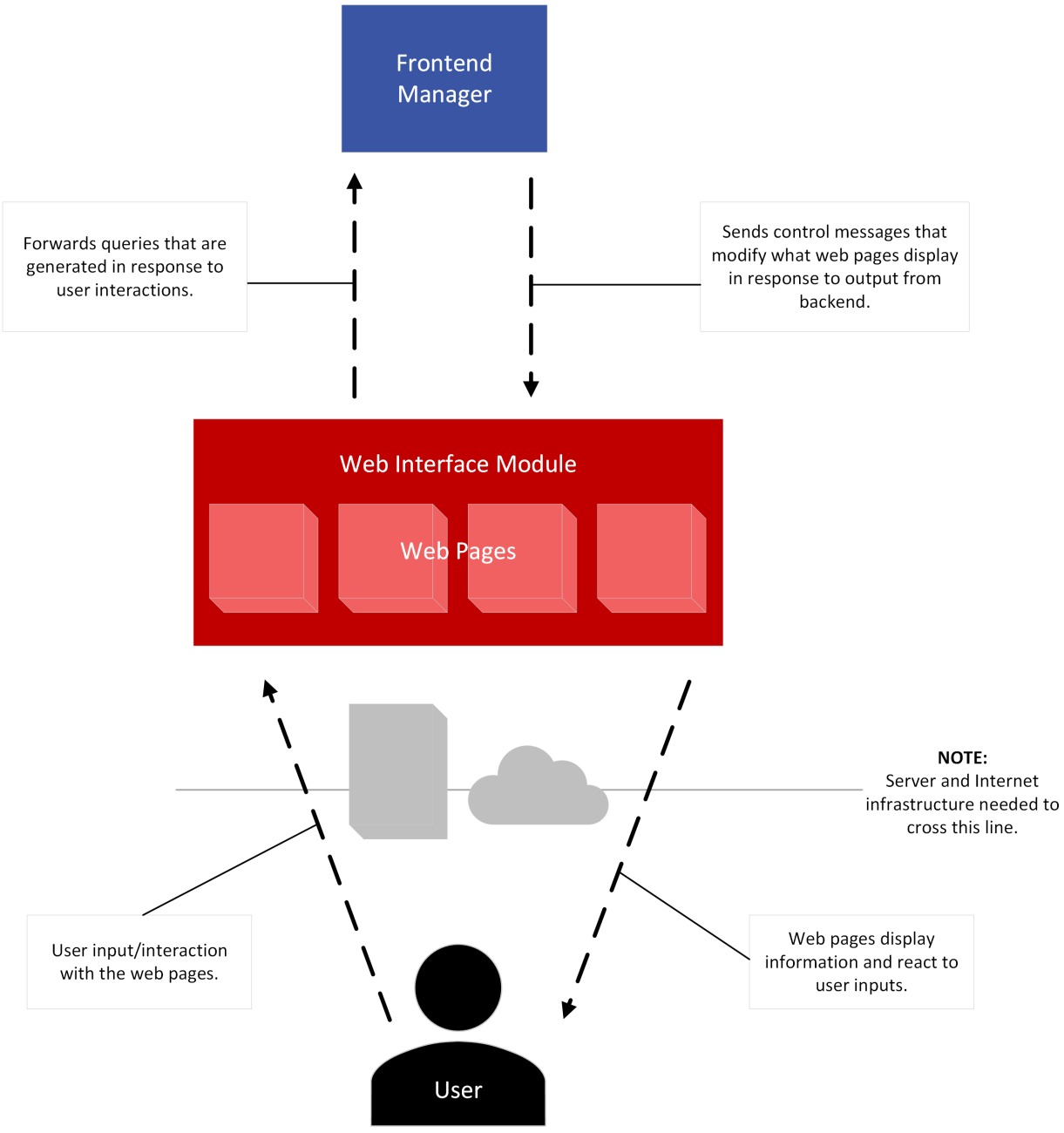

Fig. 3.4: Web Interface Dynamic Model

3.2.1 Main Page (Upload an Image)

This is the page where users can upload an image to have its dominant color extracted and related colors and palettes generated for that dominant color. For more information on the dynamics of this use case see [link to use case #1]. At first the page will only have a skeleton with blank palettes and color blocks, but after the user uploads a valid image, those blocks will be populated with the generated colors. The users will be able to filter the palettes that have been generated by selecting tags that will be displayed above the palettes section.

3.2.2 Example Palettes Page

This page of our website shows a variety of example palettes so that users can get ideas and inspiration for their own color palettes. When the user opens the page, the system will query the backend database which will result in a series of 4-color palettes being displayed as blocks. Like the palette section on the main page, the example palettes section will have several tags along the top which users can select to filter out the results displayed. For more information on the moving parts of this process see [use case #2].

3.2.3 Color Theory Page

This part of the website is purely informational. It will provide users with basic knowledge of color theory and show how the principles of this discipline have been applied in Gbiv to generate new colors after an image has been uploaded.

3.2.4 About Us Page

Like the color theory page, the "About Us" page has little to do with the dynamics of Gbiv, rather it exists to provide background to the users. Information about the project and the team are important from a developer's perspective because we like to get credit for our hard work. However, it also benefits the user because it provides an avenue for contacting the team to report bugs or to become a contributor themselves if we make this system open source in the future.

3.3 Backend Manager

The functionality of the backend manager is very similar to that of the frontend manager (see Section 3.1) in that it is middle ground for communication throughout the system. It is a vital part of the overall framework of Gbiv because without it the connection between the front and backend would be much more complex and vulnerable to bugs. Like the other "manager" module, this component is mostly defined by its interaction with other modules. However, we can still make a basic static diagram that shows the structure through which information flows:

The backend manager has both inputs and outputs from the front and backend. On the frontend, the inputs come mostly in the form of requests for data and/or computation that requires backend modules. The outputs to the frontend are entirely control messages because the backend modules that manage computation and data retrieval can return directly to the frontend manager. The outputs to the backend come in the form of function calls to either the color analyzer or database interpretor modules. In addition to these function calls, control messages may be passed along to the backend for special cases such as error handling and application updates. To show all of these inputs and outputs in a concise manner we can build a dynamic model for the backend manager:

This module was designed with ease of communication as the main goal. By establishing a central module where communication from the frontend to the backend passes, we are able to reduce the structural complexity of the system and do more with less function calls. Furthermore, by having the color analyzer and database interpretor return directly to the frontend, we avoided the need for extensive data processing and reformatting.

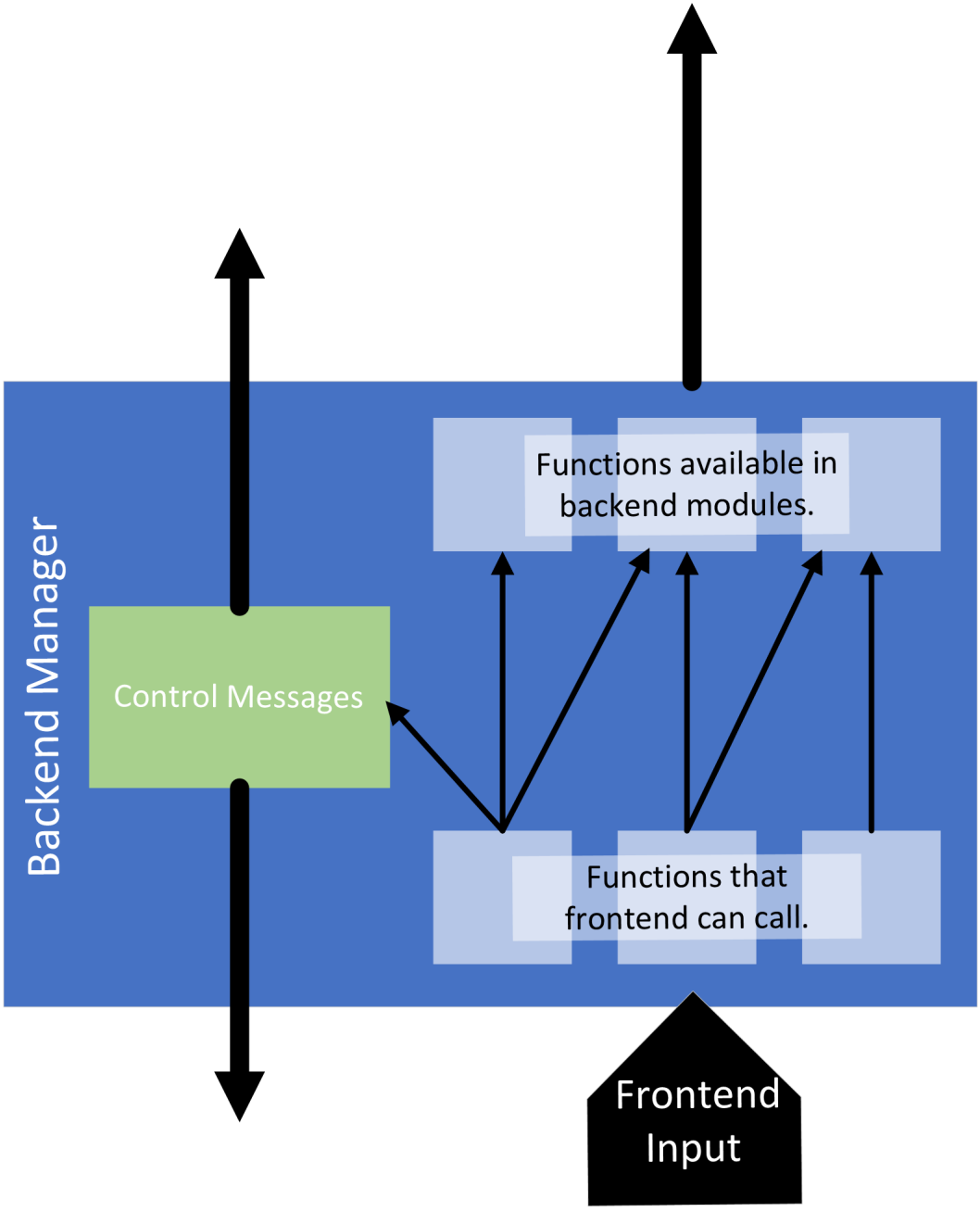

Fig. 3.5: Backend Manager Static Model

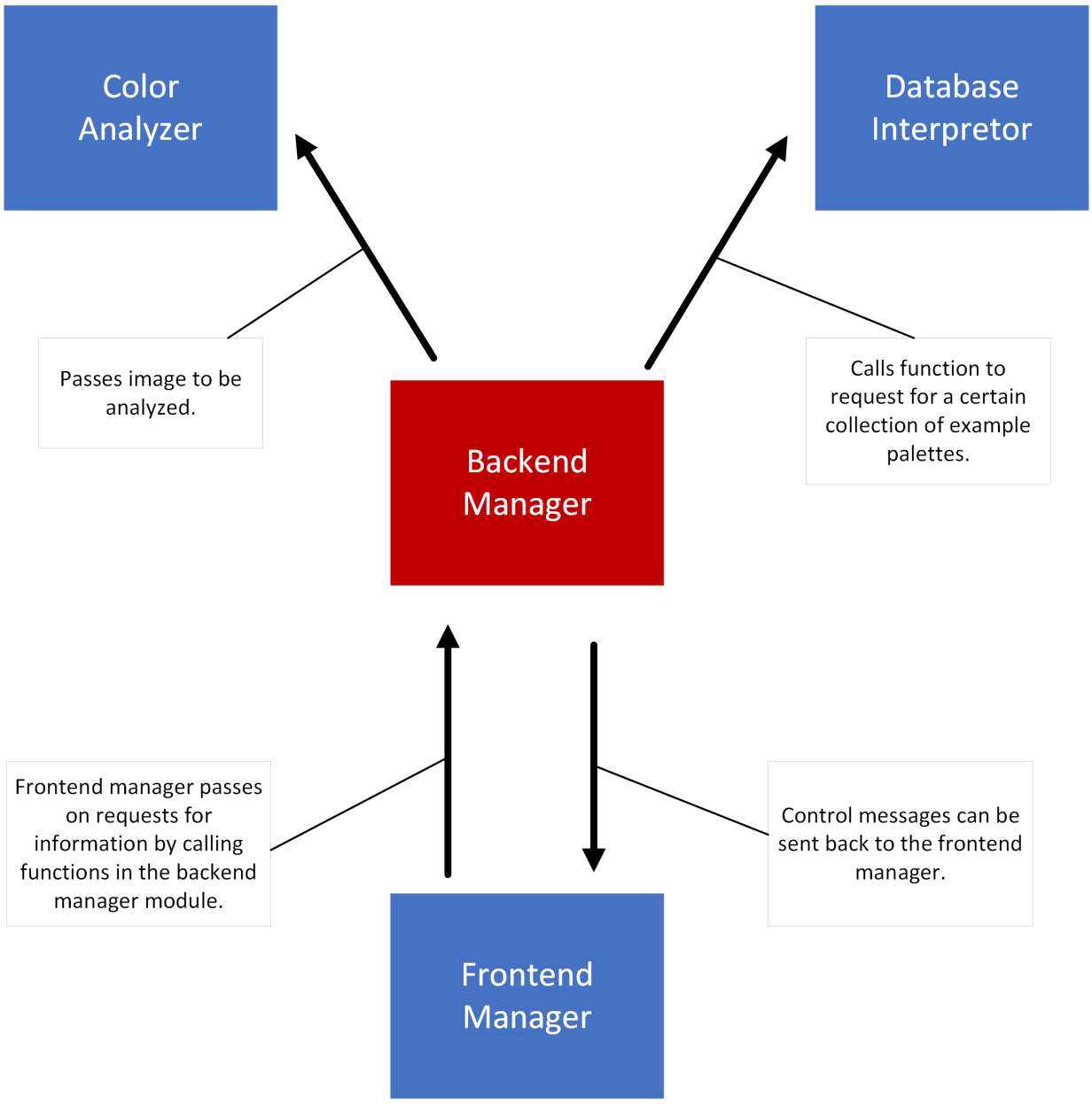

Fig. 3.6: Backend Manager Dynamic Model

3.4 Color Analysis

The primary function of this module is to the color analysis and generation that happens after a user has uploaded a photo. This module is made up of several sub-modules (divided by functionality) which are further divided into sub-sub modules. The static model below gives a visual picture of how the color analysis module is structured.

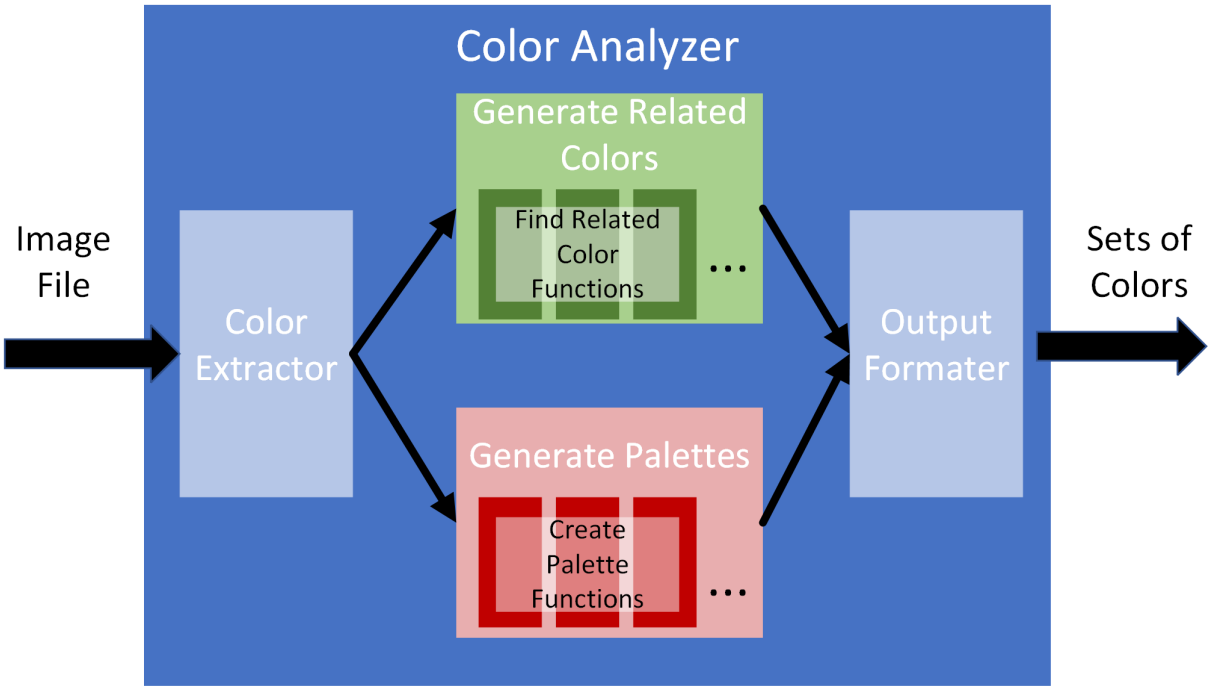

Fig. 3.7: Color Analysis Module Static Model

As the above model shows, essentially all of the work with color manipulation and analysis is done within the module. This makes for a high level of cohesion that allows for a weak coupling with other modules in the system. In fact, the color analysis module only has to interact with a single module which is the "Backend Manager." The backend manager passes an image in the form of a .png, .jpg, or .jpeg file and this module returns several sets of color codes as lists of hex code strings. The inter-module interactions of this part of the system are further specified in the dynamic model below.

This module was designed with a high degree modularity in mind. By separating the color analysis process into two parts, we are able to define two classes of sub-functions that share common features: palette generator functions and related color finder functions. This allows for code re-use and also source code that is easier to read and interpret. We also designed this module to have simple data types as both inputs and outputs. This allows easier integration with the rest of the system and fits well into our chosen framework (Flask).

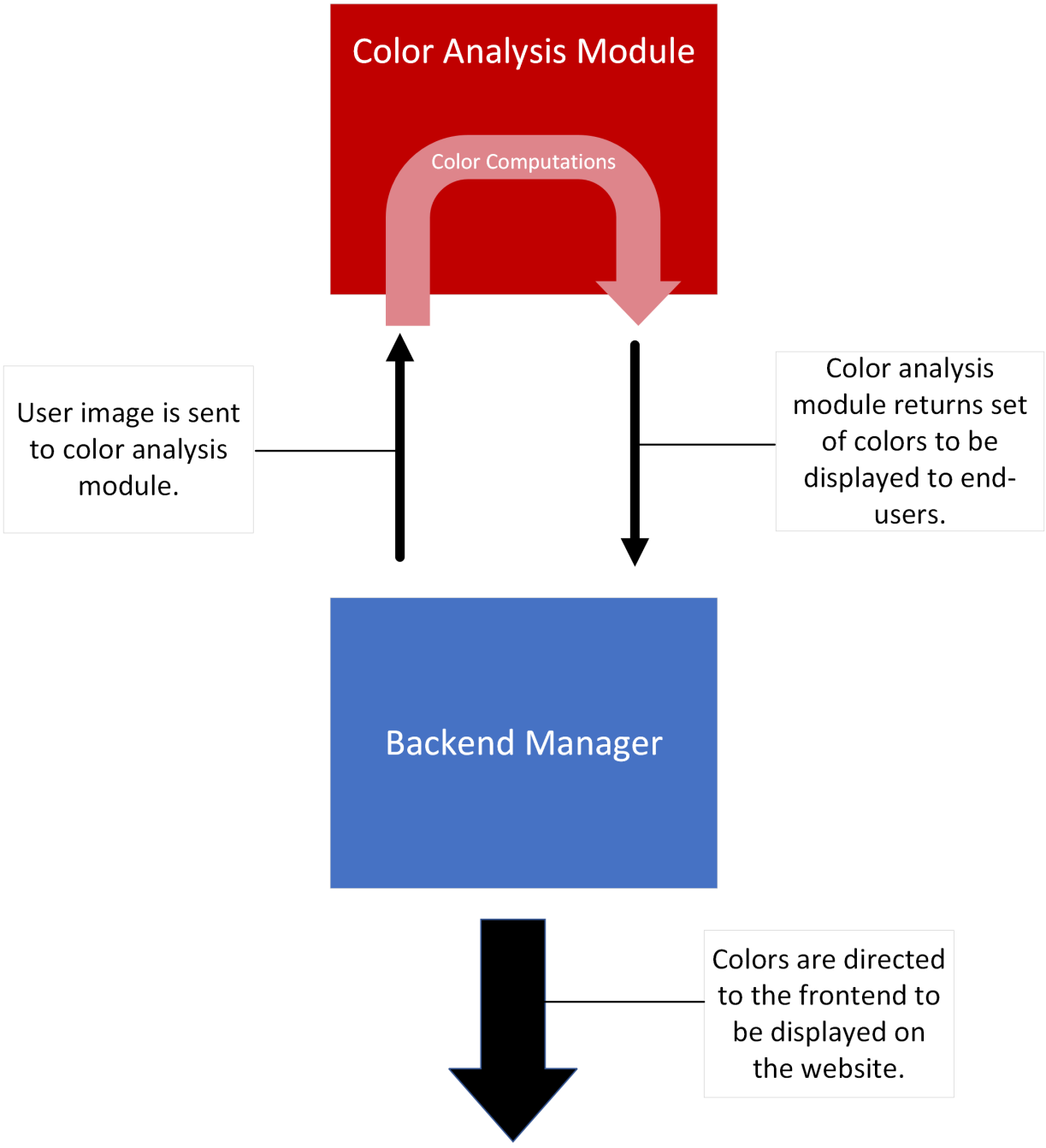

Fig. 3.8: Color Analysis Module Dynamic Model

3.5 Palette Database

The purpose of the Palette Database module will be to store popular palettes that many visitors to Gbiv have looked at. This will allow users to view a range of different color combinations and get inspiration for their design projects. Because there is only one collection of elements in the database for this project, the design of the database itself is somewhat straightforward. The static model below shows the layout visually:

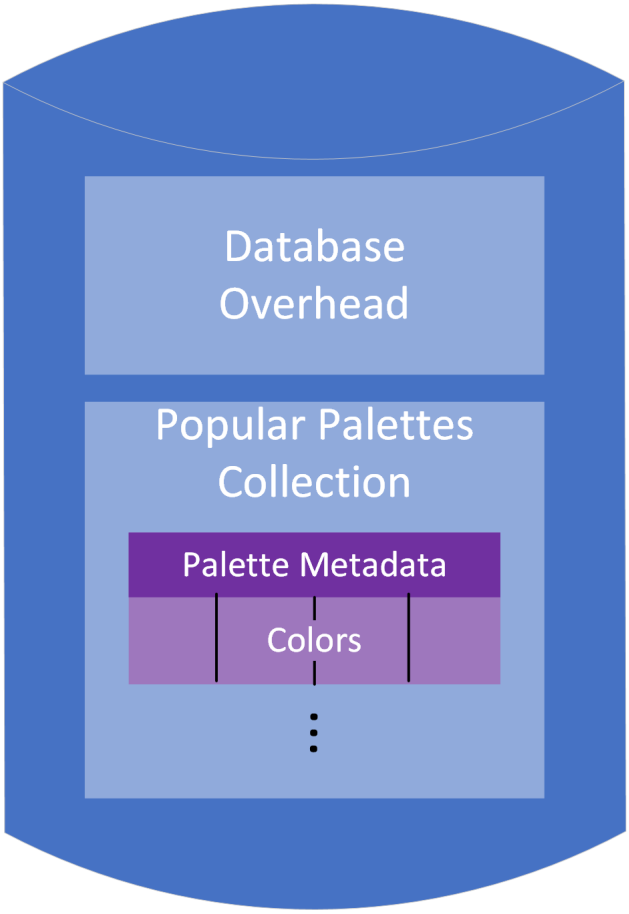

Fig. 3.9: Palette Database Static Model

The technology we will be using to implement our database (MongoDB) comes with a library that allows for efficient interfacing through python. Because of this built in advantage, we have designed the system so that the database has one module with which it communicates, the "Database Interpretor." This interface consists of a single type of input and a single type of output. When a user visits the "Popular Palettes" page, the frontend will query that backend which will reach the palette DB as a request to view the collection of palettes. When this query happens, the database will pass the collection on to the database interpretor module in a format that allows for easy movement to the end-users. Below we have included a dynamic model to demonstrate this interface.

The design choices for the palette database module were made with the goal of simplicity. By keeping the number of collections to a minimum and formatting all data entries identically, the organization and movement of Gbiv's data can be straightforward and efficient. This prevents database accessing from being a bottleneck for performance, as well as reduces the need for more modules for data formatting.

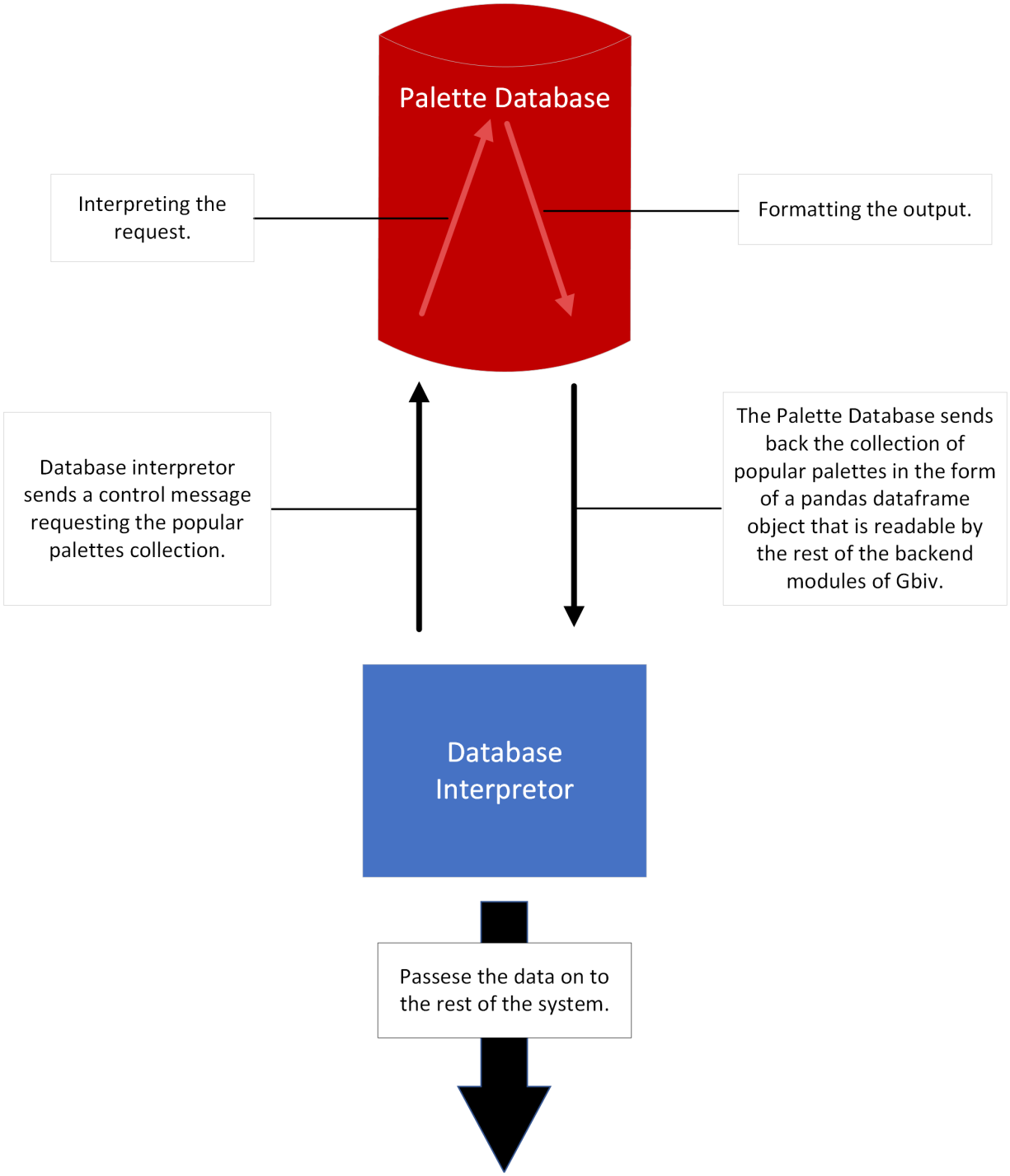

Fig. 3.10: Palette Database Dynamic Model

3.6 Database Interpretor

The database interpretor module exists so that the palette database can interface with the rest of the system. MonogDB was used to implement the database and it can be controlled by a python script using provided libraries. The rest of the system has also been implemented in python (with Flask as a framework), so it is very important that the Mongo database communications are translated into a python context. The database interpretor fills this need. To get a better idea of the structure of the interpretor, we can visualize it with the static model below:

Because of the nature of the this module, its interface with the rest of the system is defined by the needs of the palette database. Specifically, the database interpretor receives inputs from the backend manager which ask to query the DB for a certain collection of palettes–the interpretor translates these queries into a format compatible with MongoDB and sends them to the palette database. The database then returns collections of palettes to the interpretor, which again reformats the input and sends it as an output to the frontend manager. These interactions are more clear when shown visually as in the dynamic model below:

The idea behind having a module devoted entirely to interfacing with the database is that the technologies behind data storage and the rest of the system are very different. By using an interpretor, we can more smoothly integrate the example palettes into the end-user experience and have a more cohesive system overall. From an implementation point of view, this lets us code in python as much as possible which helps to reduce bugs and maintain consistent style/organization.

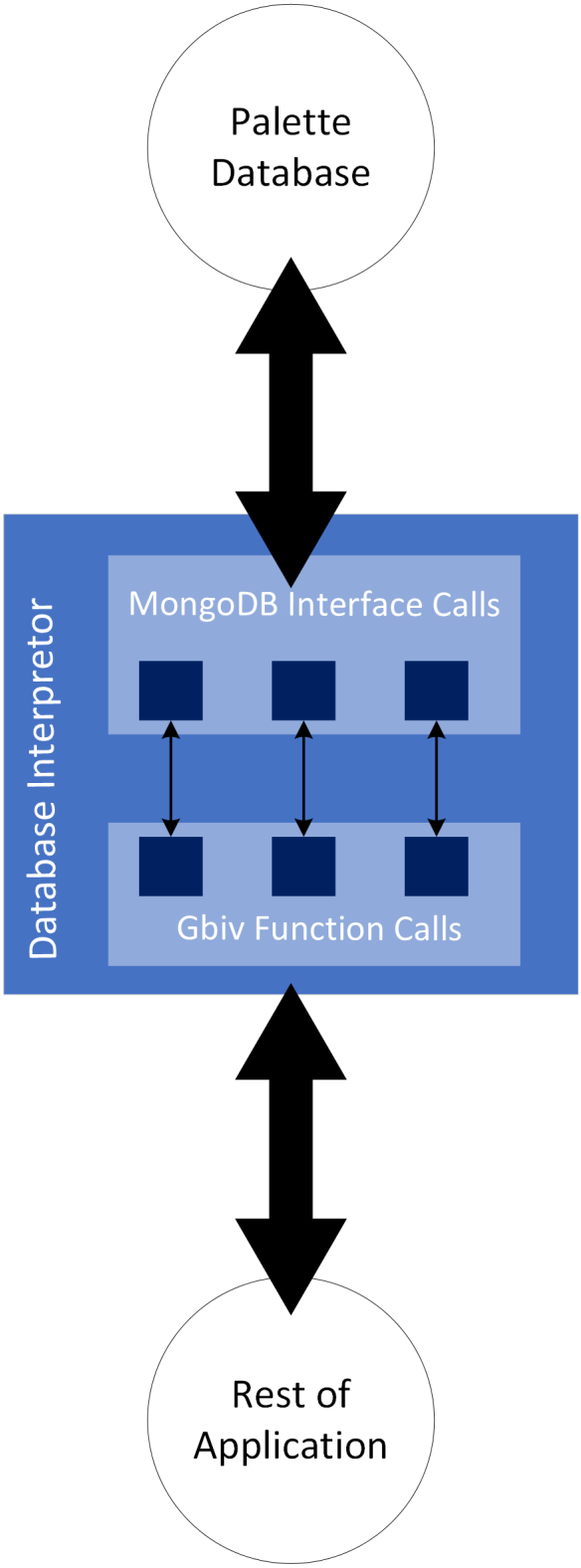

Fig. 3.11: Database Interpretor Static Model

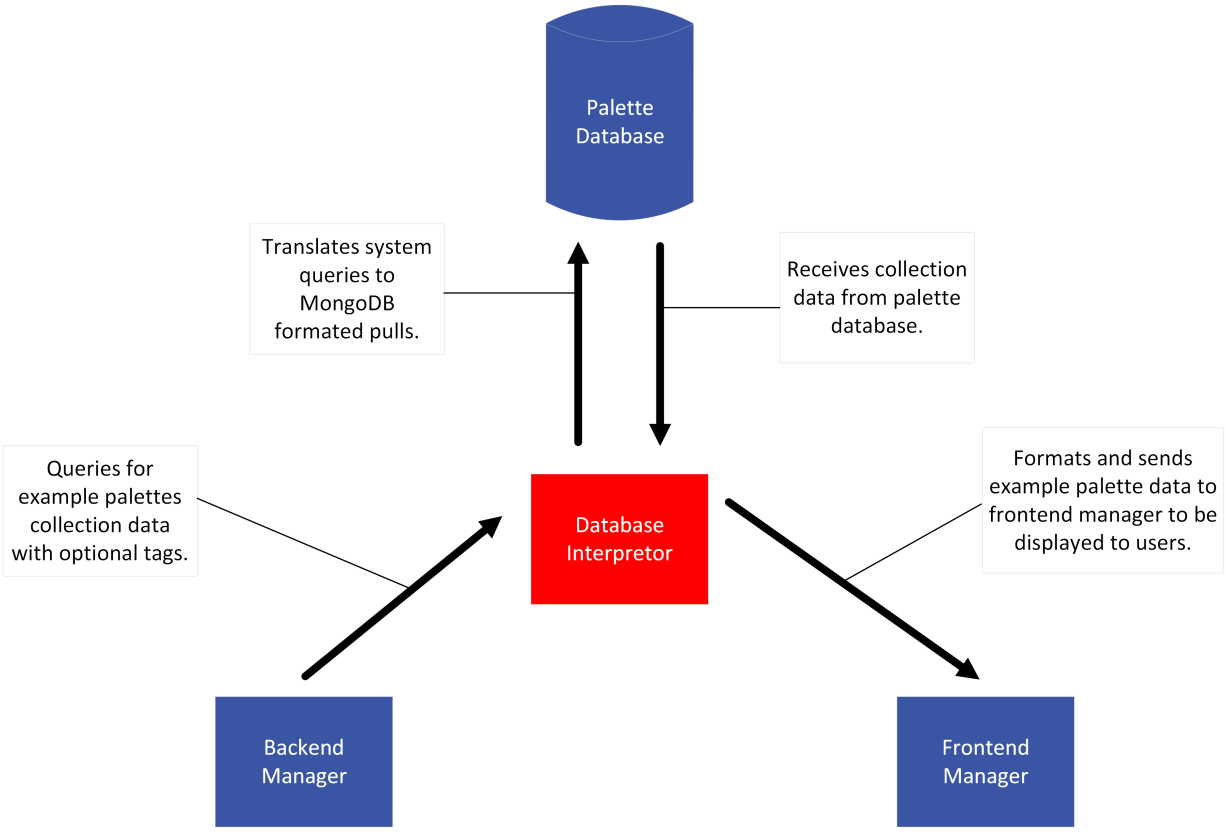

Fig. 3.12: Database Interpretor Dynamic Model

CHAPTER

FOUR

DYNAMIC MODELS OF OPERATIONAL SCENARIOS

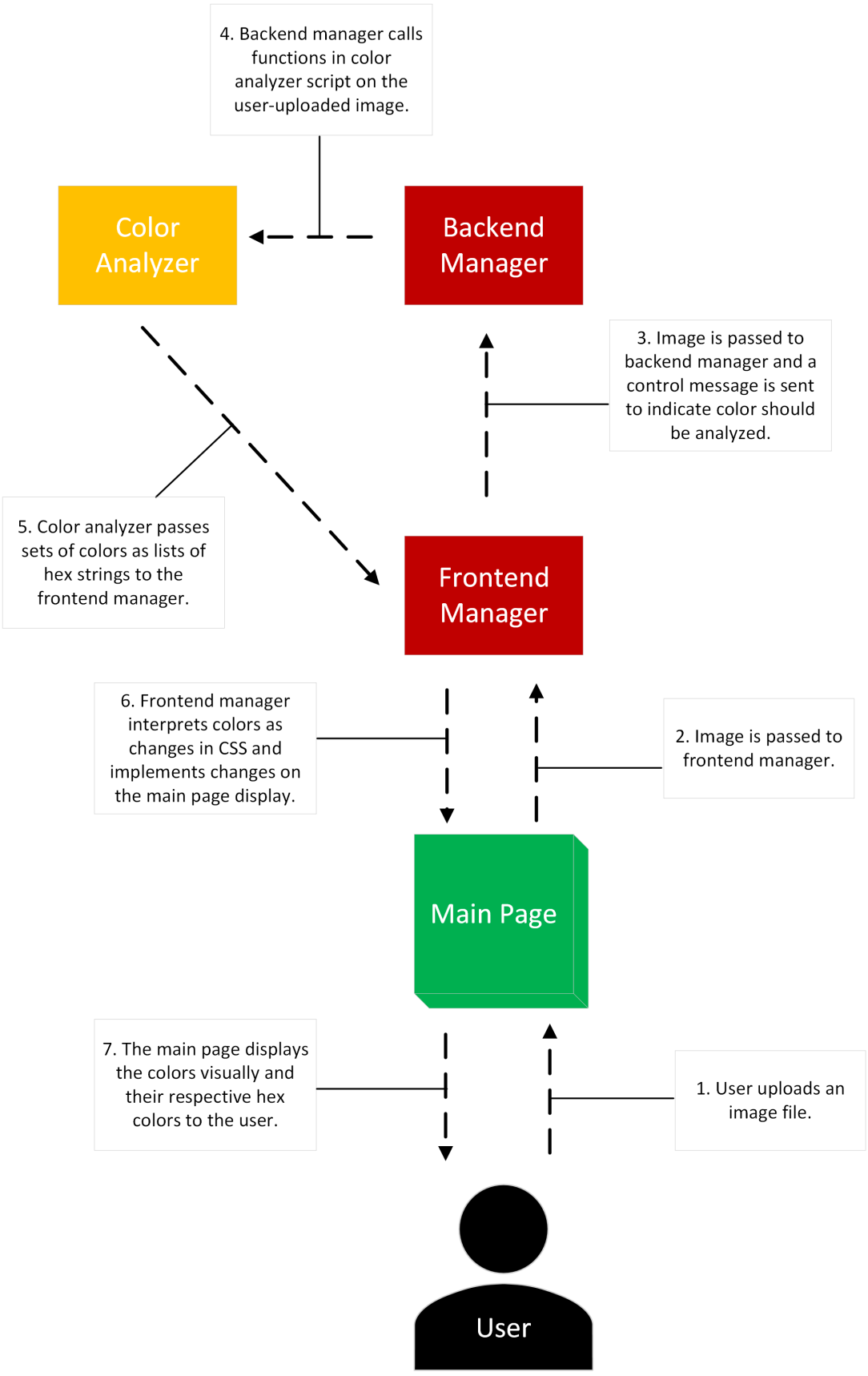

Fig. 4.1: Use Case #1: Uploading an Image for Color Analysis

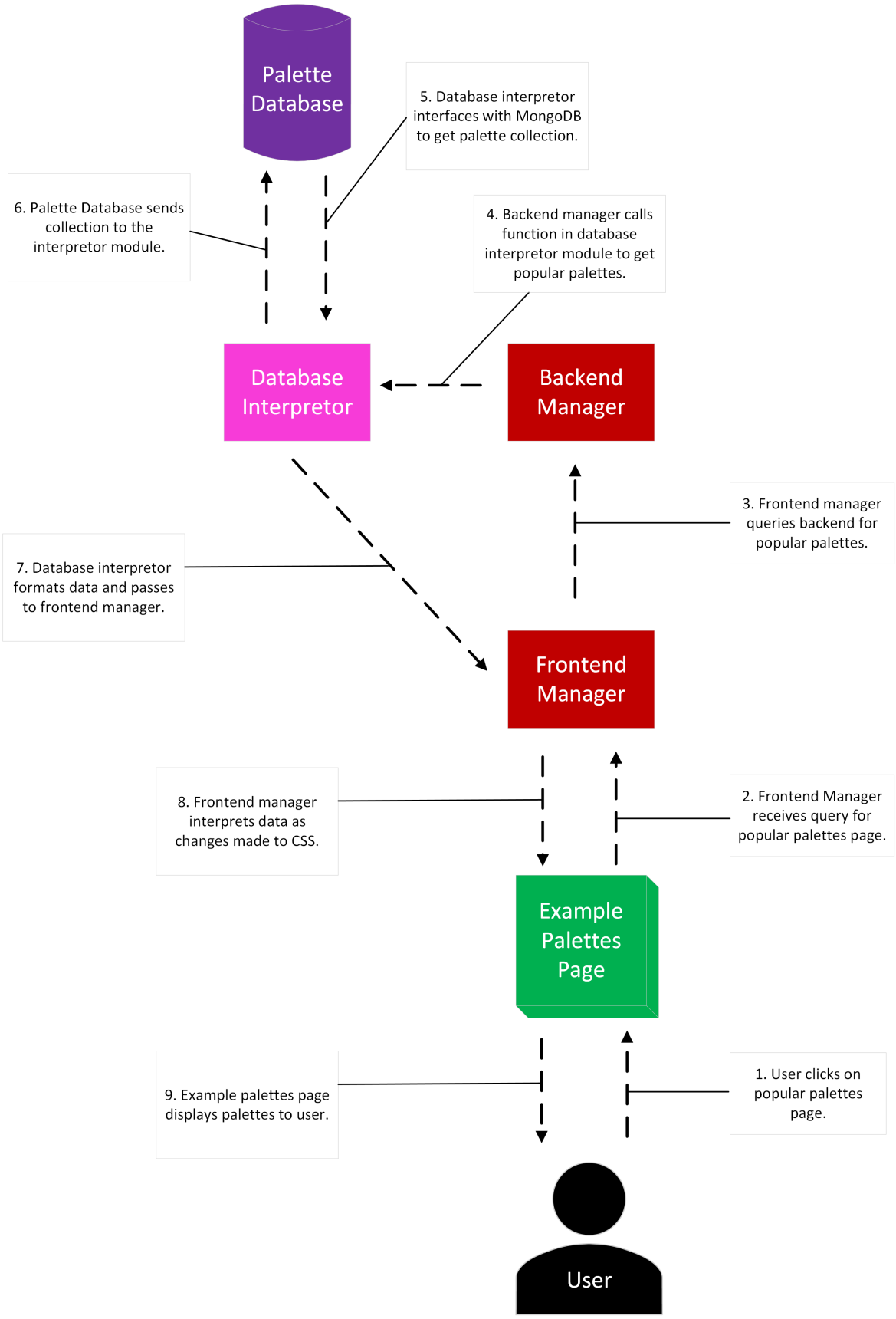

Fig. 4.2: Use Case #2: Displaying Example Palettes

CHAPTER

FIVE

ACKNOWLEDGMENTS

The format of this SDS document was originally created by Professor Juan Flores. Reference to a completed SDS document was provided by Ronny Fuentes, Kyra Novitzky, Jack Sanders, Stephanie Schofield, Callista West with their Fetch project SDS.