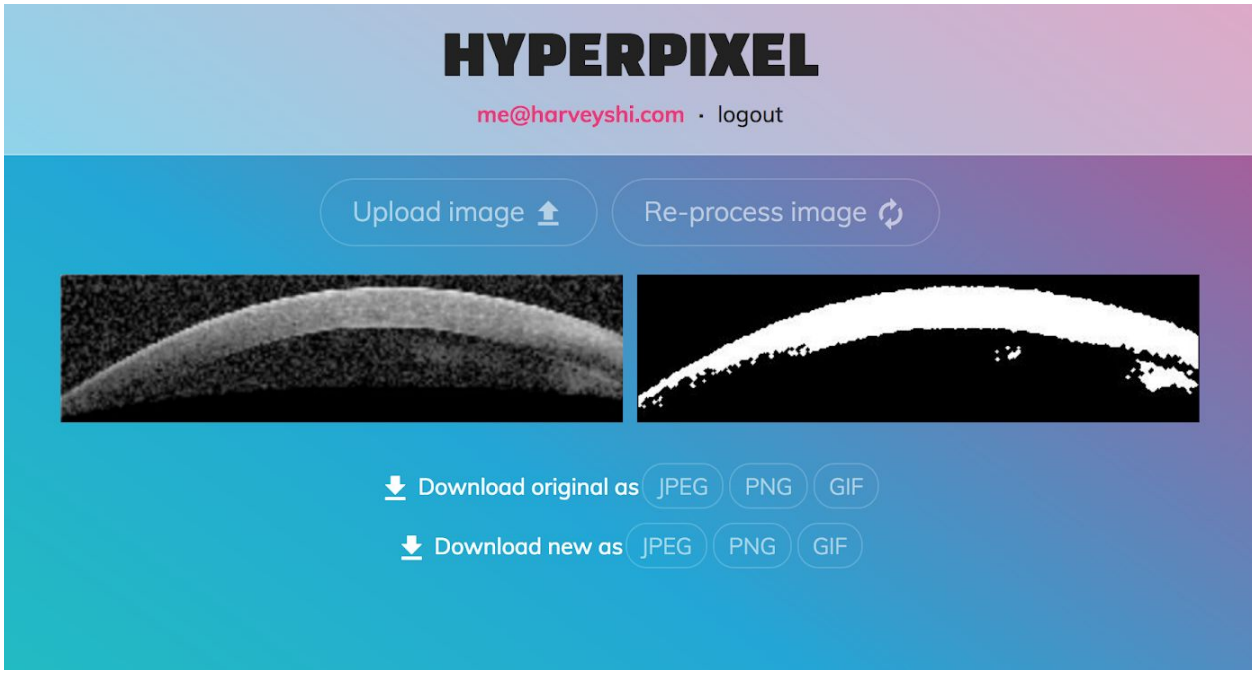
HYPERPIXEL

Project RFC



Authors

- Harvey Shi - harveyyshi@gmail.com
- Michelle Wei - michellewei12345@gmail.com
- Edward Liang - ed.l.1324@gmail.com

Reviewers

- Mark Palmeri - mpalmeri00@gmail.com - PENDING
- Suyash Kumar - suyash@suyashkumar.com - PENDING

Links

- Github Repo - https://github.com/rvshi/ImageProcessorS18
- Frontend Website - https://harveyshi.com/ImageProcessorS18/
- Demo Video - https://youtu.be/1ffBEW7qsT0

Table of Contents

Abstract

Image segmentation is a critical component of many scientific workflows. Our goal is to design and implement an online image segmentation service that provides an intuitive frontend interface and RESTful API. We designed and built an API backend using Flask and replicated via Gunicorn. A MongoDB database was used to save user information and links to images cached on the server. The entire backend and database are run in production using Docker containers and managed via Docker Compose. We also created a static frontend interface for interacting with the API, built with React.

Background

Segmentation is usually the first step in any type of automated image processing. It is often used in recognition, detection, and measurement of objects in images. Segmentation subdivides an image into different regions or components that correlate with features of interest. It typically groups pixels in the image that have similar attributes, such as intensity, hue, range, texture, etc. Some examples of its use can be found in industrial inspection, character recognition, object tracking, motion correction, and anatomical feature detection in medical images.

Our lightweight API is a tool for users to quickly and efficiently carry out segmentation on their images. We can demonstrate its usefulness for segmentation of corneal layer boundaries. This can provide information about the curvature and thickness of layers in the cornea which is important for clinical procedures such as refractive surgery. Additionally, collection of these data is useful for the diagnosis and study of anterior segment diseases.

Frontend Design

The frontend was designed for simplicity and ease of use. It consists of two pages:

1. Login: on this screen, users can input their username and password (Fig. 1). The circle on this page is designed to evoke an eyeball shape.
2. Dashboard: this screen displays the original and processed images and handles the various API actions such as uploading, processing, and downloading images (Fig. 2).

The styling was executed using custom-written CSS and is fully responsive for use on mobile devices (Fig. 3).

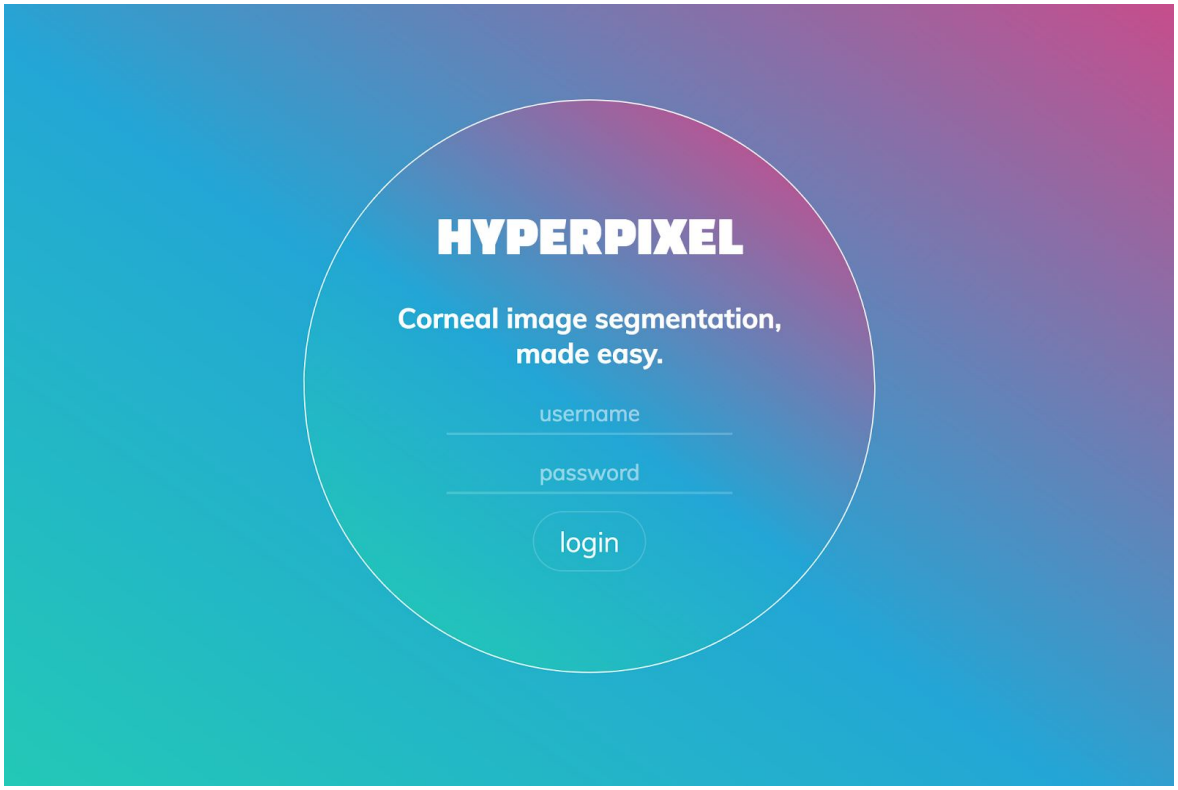


Figure 1. The login page includes fields to input a username and password.

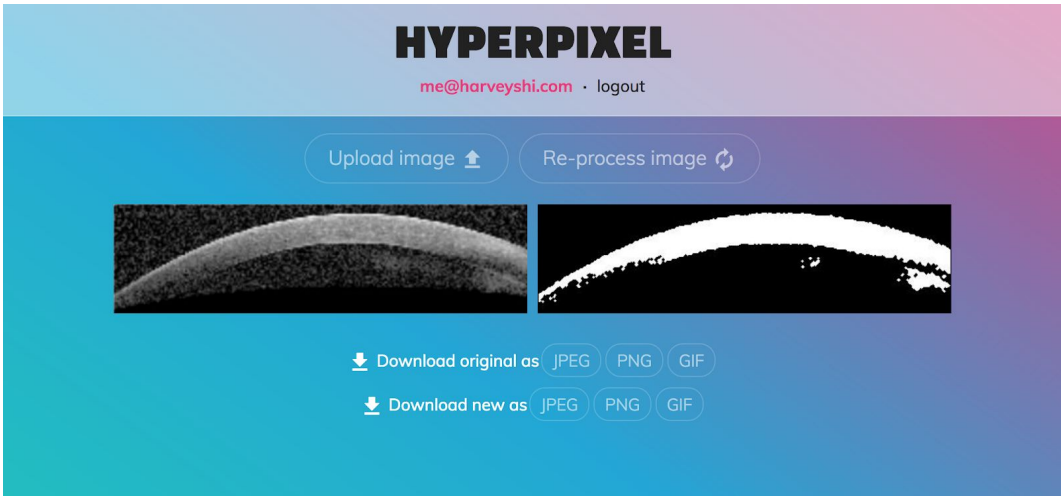


Figure 2. The dashboard page displays the original and processed images. It also includes options to upload a new image, reprocess the current image, and download either the original or processed images as JPEG, PNG, or GIF files.

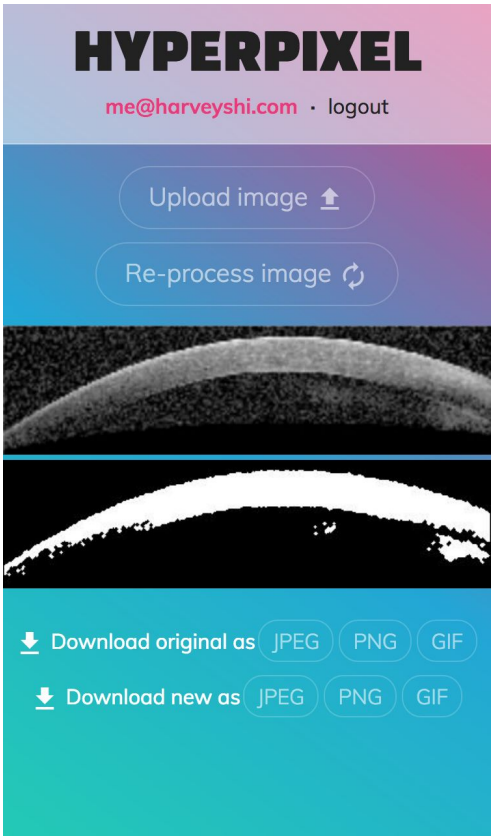


Figure 3. A screenshot of the frontend interface running on a mobile device, demonstrating the responsive styling.

Architecture

The entire backend is running on a Duke Ubuntu 16.04 VM and consists of a Flask server and MongoDb database. The Flask server enables the image segmentation operations and API endpoints. Flask is replicated for production via gunicorn. The server also stores and retrieves user information via our MongoDB database.

The backend and database are running in their own Docker containers, which are linked and executed via Docker Compose. Through the container linkage, port 27017 of the database is exposed only to the container for the backend. The backend container exposes port 5000 to the host machine. Incoming traffic to port 80 of the server is routed to port 5000 through an NGINX proxy, which is SSL encrypted via a certificate from letsencrypt.org.

The frontend interface implements requests for all the API endpoints, but the API is also designed to be used without any GUI. See Tab. 1 for more detail on the specific endpoint formats. Fig. 4 depicts a high-level overview of the architecture details.

A user can login with their username and password only if they have been added to the database, which is currently done manually. This is done via a POST request with the login details to the /login endpoint. If the user exists and they have entered the correct password, a JSON Web Token (JWT) are returned for the user to be added to the header on all subsequent authenticated requests. There is also a /validate GET endpoint, for checking if a given JWT is still valid. This is useful for the frontend, when verifying if the user session is still active upon page reload or returning to the site.

Via the /upload POST endpoint, the user can upload an image in PNG or JPG format, which is sent as a base 64 string. Once on the server, the Flask backend saves the image string as a file on the virtual machine (VM) server. The image is also assigned a Universal Unique Identifier (UUID), which is stored in the Mongo Database and can be used to retrieve the original base 64 image using /download.

On the user's /process POST request, the original image is segmented to generate a new, processed image that is also stored as a base 64 string in the VM server and assigned a new UUID. Segmentation is done through application of several SciPy module functions: Otsu thresholding, binary opening to remove small objects, and binary closing to fill small holes.

The user can retrieve either the original or processed image through the /download POST request, which takes in the file UUID and the filetype to convert the image to (JPEG, PNG, or GIF). This endpoint is used in conjunction with the /list GET request to display both the original and processed images on the Frontend.

Diagram

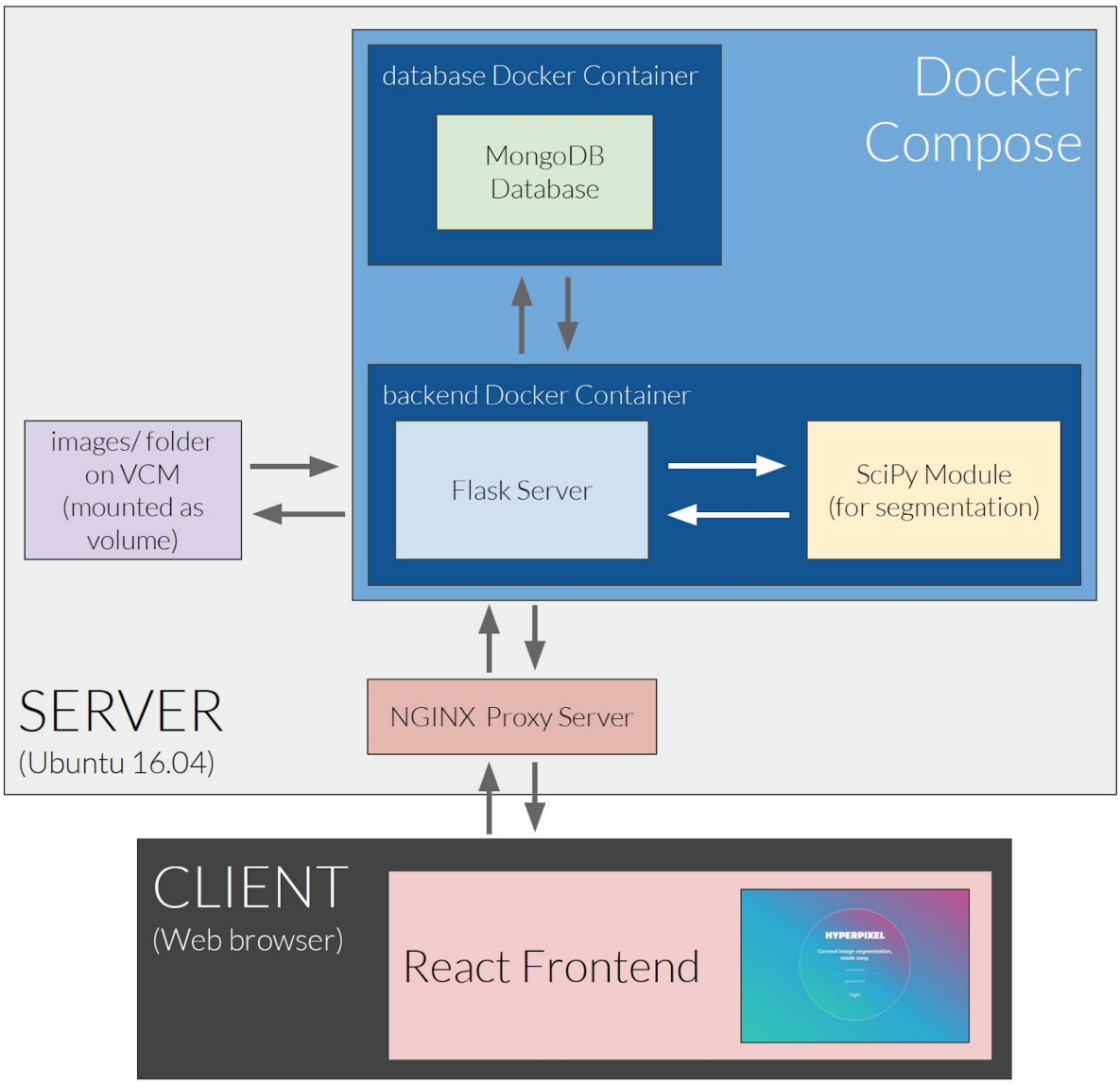


Figure 4. Diagram of the interactions between client frontend and server backend: the user interacts with the React frontend, which communicates with the Flask server. The Flask server implements actions such as storing and retrieving images by communicating with the MongoDB Database and the VCM server. It also uses the SciPy processing module to segment the images.

Endpoints

Table 1: The following table describes the endpoints and usage available in our API.

Endpoint	Method	Description	Request	Non-Error Response

`/login`	**POST**	Log in to service	`{`
			`username: <valid email>,`
			`password: <password>`
			`}`
				`{`
				`jwt: <JSON Web Token>`
				`}`

`/validate`	**GET**	Check if JWT key is valid	`JWT in header`	`{`
				`username: <email of user>`
				`}`

`/list`	**GET**	List the images for a specific user	`JWT in header`	`{`
				`originalID: <uuid of original image>,`
				`processedID: <uuid of processed image>`
				`}`

`/upload`	**POST**	Upload new image to database	`JWT in header`
			`{`
			`username: <valid email>,`
			`file: <base64 encoded string>`
			`}`
				`{`
				`fileID: <uuid of image>`
				`}`

`/process`	**POST**	Segment current stored image	`JWT in header`
			`{`
			`username: <valid email>`
			`}`
				`{`
				`file: <base64 encoded string>`
				`}`

`/download`	**POST**	Download image	`JWT in header`
			`{`
			`username: <valid email>`
			`fileID: <uuid of image to download>`
			`filetype: <jpeg, png, or gif>`
			`}`
				`{`
				`file: <base64 encoded string>`
				`}`

Database

We are using a MongoDB database, and the schema includes fields for user login information, and uuids of uploaded and processed images (Tab. 2). The images themselves are stored in the virtual machine.

Table 2: schema for User entry in database.

Field	Field type	Description
`username`	EmailField	Unique user email
`password`	CharField	User password
`original_image`	CharField	Image UUID that points to an image stored in the VM server
`processed_image`	CharField	Image UUID that points to an image stored in the VM server

Infrastructure

Our backend service can run on any computer with Docker and Docker Compose installed. This could include virtual machines or private development servers. Currently the service is running on a Duke VM, with the static frontend being hosted on Github Pages.

Security Considerations

Our service does not handle sensitive patient information, and does not rely on any proprietary secrets. The database currently contains a few allowed users with unique emails and associated passwords. Only the most recent original and processed images per user are stored in the database. Access to the uploaded images and processing capabilities requires authentication via a JWT obtained through the /login endpoint. Currently, the frontend stores the JWT in localStorage, which could potentially be vulnerable to cross-site scripting (XSS). As such, both the frontend and backend use HTTPS and TLS encryption.

Failure Modes & Mitigation

The frontend is statically served from Github Pages, which could fail due to issues with Github's servers. There is not much that could be done to mitigate this possibility, with the exception of using our own server to host the frontend.

The backend service relies on a single Duke VM to serve the app. Should the server shut down unexpectedly, users will be unable to access the API for a short while. After the server reboots, the Docker Containers for the backend and database will automatically restart and the service should be back to normal.

Users also might upload image files that are too large or the wrong format. This is addressed by having a max file limit of 4 MB imposed both by the frontend and by the NGINX proxy. The format is handled by the frontend.

Alerting & Monitoring

The backend service will print to a backend.log file stored on the VM. This file can be accessed by the Docker Container since the backend/ folder is mounted as a volume. Users will not receive any alerts external to the website (emails, push notifications, etc.). However, small notifications will appear to the user on the website on successful login, logout, and processing. Alerts also appear when errors occur during login or processing, e.g. the image cannot be uploaded because it is too large.