CMPT475
Capping Project Paper

Kai Wong (CS), Antonio Delvecchio (CS), Zachary Recolan (CS), Brendon Boldt (CS), Leonardo Keefe (IT)

Table of Contents

Introduction

For our capping project this semester, we worked on an Artistic Stylizer Platform (ASP), which allows users to artistically enhance photos uploaded to our server. The enhancement process uses deep learning to analyse the artistic style of an image and apply it to an uploaded photo or video. This will allow the user to manipulate their images or videos in a way that truly empowers their artistic freedom. Many platforms allow users to manipulate their media, however very few will be able to offer the ability to upload “style” images of works from famous artists like Van Gogh or Picasso to manipulate their media in an endlessly flexible way.

Analysis

User Requirements

1. The user must be able to upload an image or video to an online platform and have it “artistically enhanced”
    a. We will use existing deep learning code base can be thoroughly understood and modified for the purposes of this project as long as it is available under the Torch license
    b. 100 People must be able to use this system at any given time
    c. The System will have an anticipated growth to 1 million users in 2 years
    d. An iOS app is not necessary, but a mobile responsive website is
    e. A server with a GPU will be available for styling the photos and retraining the network on new styles
    f. This will be different and special from similar platforms, as the website may be used for free, and there is an additional API service offered to paid users
        i. The ability to upload custom styles and the ability to retrain the deep learning engine on user command are also distinctive features
    g. Users will have to create an account to use the platform
        i. An account will allow the user to upload images to be styled with a given selection of styles
        ii. Users can pay an amount of money to upgrade their account and receive benefits discussed in a later section
    h. Users can only upload one image at a time
        i. Picture formats: JPEG, PNG
            1. Size constraint for images: 7MB
        ii. Video format: mp4

2. This artistic enhancement process involves the transferring of the style of an artistic image to their uploaded image
    a. Free users
        i. There will be a predefined style list for free users

        i. Free users will receive an unobtrusive watermark on their enhanced images
        ii. A free user may style as many images as they please
        iii. A free user may only store two photos on the platform
    c. Paid users will benefit from the ability to store more images on the website/system, the use of the API, and no content watermarking
        i. Paid users will also have the ability to style videos
        ii. Paid users will also have the ability to upload their own style images to style their content with

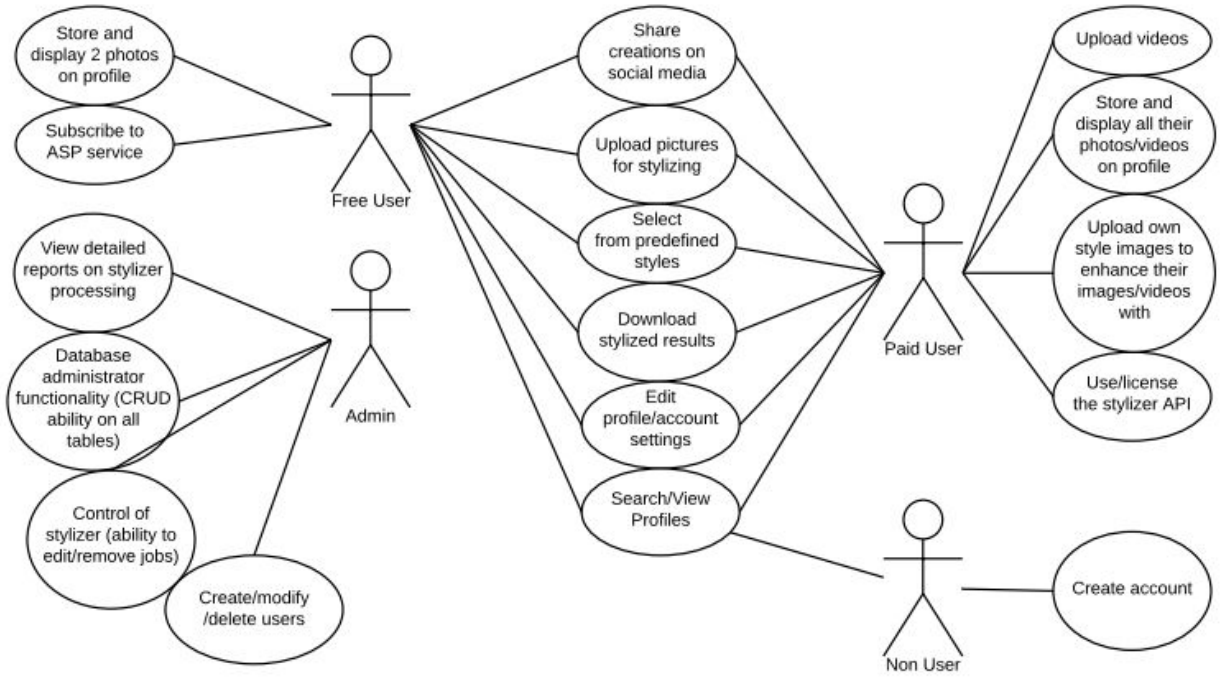
3. The artistic stylizer platform will have 3 components
    a. Stable web platform that can scale to serve a growing amount of users
        i. Compatible with LAPP(Linux, Apache, Postgres, Express) server
        ii. Website should look flawless on mobile and desktop
            1. Use valid HTML5 and CSS3
        iii. Security is important
            1. Non-authenticated users should not be able to access any data through site
                a. They should still be able to view other user profiles
    b. Deep learning backbone that performs the image enhancement services
        i. Allow a style to be transferred to an image after it has passed through a queue of images processed from at least 100 concurrent users
        ii. Algorithmic selection to create high quality images in a manner that is not computationally excessive
    c. Data store that allows users to store credentials/create a profile
        i. Users must be able to create a profile to save their images and styles
        ii. User sharing capabilities to social media
        iii. Security is a concern here due to projected growth
            1. Users must be authenticated to view website data, authentication credentials will be hashed with SHA256
        iv. Images will not be purged from the file system and database automatically after a given period of time
            1. They will instead be purged based on the number of images allowed for their free or paid license and storage quota

4. Must have the ability to create admin users
    a. Must be able to receive a report of system usage by ranges of dates
        i. Usage must be metered as data is uploaded and processing time is spent
    b. Admins must be able to view a queue of current processes

5. Tool must be available to imaging/recording/picture studios across the world via licensing
    a. The ASP admins will maintain full control of the database and infrastructure

b. Requires an API that allows a user to
    i. Authenticate credentials
    ii. Send media
    iii. Send parameters
    iv. Receive media
c. Users will use a private cloud as the public one is not secure
    i. Must meet all of the DOD Standards
        1. Implementation of DOD and NIST security standards is critical
        2. All personal info stored in hash (SHA256)
    ii. Full report tests must be done in respect to the security of the cloud

Use Case Diagram



Plan

Project Plan

8/30/2017		Week 1
	Form team: make introductions; inventory skill strengths and weaknesses; formulate and document initial roles and responsibilities	
9/6/2017		Week 2
	Brainstorm client questions and project requirements	
9/13/2017		Week 3
	Develop and finalize project plan	
9/20/2017		Week 4
	List of IT requirements	
9/27/2017		Week 5
	Build a postgres database	
	Complete and submit 5 mock-ups (ie. wireframes) of the early demo views of the user interface	
	Deep learning output using Google's pretrained network	
10/4/2017		Week 6
	Deep learning style transfer prototype	
	Continue to build website (no functionality)	
10/11/2017		Week 7
	Prototype of database submitted for evaluation	
	Preliminary website built (no functionality)	
10/18/2017		Week 8
	Setup connection between database on server and UI on front-end	
10/25/2017		Week 9
	Implement profile functionality	
	Implement login functionality	
	Implement web APIs to access database	
11/1/2017		Week 10
	Final DB Design	
	Database Load Tested	
	Implement web API to call deep learning component	
	Hook up deep learning component to everything else	
	Fully functional deep learning style transfer	
11/8/2017		Week 11
	Implement other functionality (styling)	
	Implement other functionality (photo uploading)	

	Stylizer Queue Load Tested	
	Help ensure all functions are working as intended	
	Finalize UI design and submit	
11/15/2017		Week 12
	Continue work on implementing/improving existing site and backend functionality	
11/22/2017		Week 13
	Continue work on implementing/improving existing site and backend functionality	
11/29/2017		Week 14
	Continue work on implementing/improving existing site and backend functionality	
	User validation tests to be performed by client documented	
	Complete project prototype first-pass demo ready	
12/6/2017		Week 15
	Prototype Deep Learning Engine Complete	
	Prototype Admin Tools Complete	
	Prototype Website Complete	
	Prototype Database Compete	
12/13/2017		Week 16
	Fully complete project	
	Final Trained Deep Learning Network	

Our plan was to build the website and UI quickly without much conceptualization, which led to some disagreement on the design and a major restructuring in design. This made us meet the October 11th “Preliminary Website Built” deadline by a much narrower margin than originally expected. We also set out a bit ambitiously as far as the deep learning aspect of the project went. Originally, the process was going to be built from the ground up, i.e a neural network in the TensorFlow engine from scratch. But by mid October we realized that just getting the engine ready to start training could easily be a semester long project on its own. After this, we switched between Torch implementations to find one that would fit our purposes, had adequate documentation and also could work with our original algorithm. The PostgreSQL database was relatively easy to build and implement due to the fact that our team had prior Oracle database experience. Connecting the front-end (implemented in Angular) with the database took a lot of time, requiring middleware APIs to be written using Node.js’ Express package in order for communication to take place. However, this allowed the API requirements for the platform to be fulfilled at the same time.

Project Diary

9/27 Week 5

Green	Kai	Figure out how to create a REST API
Green	Zack/Kai	Mockup creation started on and completed
Yellow	Zack/Kai	Creation of web UI modeled after the mockups created
Yellow	Antonio	Get the database on the server
Yellow	Brendon	Play with training Inception5h architecture
Yellow	Brendon	Test out-of-the-box stylizer using VGG19 architecture
Yellow	Leo	Prep Servers

Week 5 Marked the start of our actual build phase of the project; at this point we had finished gathering the user requirements (seen in the “User Requirements” Section), developing the project plan above and creating the initial IT infrastructure requirements based on the user requirements (seen in the “IT Requirements” Section). We had already taken leaps in modeling and even building the initial UI using Angular, and Kai had learned how to implement a “REST API” using Node.js’ Express. At this point we had yet to decide on a machine learning architecture in which to complete our styling and brendon was hard at work testing the Inception5h and VGG19 architectures. Access had just been granted to the vSphere servers that we used as our testing environment so these were being provisioned and spun up. The database was already designed and the creation scripts were generated; however, we had yet to initialize postgres and actually implement the database.

10/4 Week 6

Green	Zack/Kai	Creation of web UI modeled after the mockups created
Yellow	Zack/Kai	Establish communication between DB and UI
Yellow	Zack/Kai	Continue to tweak/improve/add to web UI
Yellow	Antonio	Get the database on the server
Red	Brendon	Self-train architecture (likely dead end)
Green	Brendon	Test out-of-the-box stylizer using VGG19 architecture
Green	Leo	Prep Servers for hosting

At this point the Angular website was completely modeled without functionality and the vCenter servers were ready to accept the necessary infrastructure to host the website and database. We were still in the process of figuring out Postgres - mostly in how to communicate with it using PGadmin to make our database development easier. We were also in the process of installing

required packages through Node Package Manager in order to run our applications. Both of the architectures for the deep learning in week 5 led to dead ends, as they did not fit our projects needs.

10/11 Week 7

Yellow	Brendon	Set up Express.js server in charge of styling
Yellow	Zack/Kai	Revamping UI styling/layout/design
Yellow	Zack/Kai	Implement rest of communication between DB and UI
Green	Zack/Kai	Establish communication between DB and UI
Green	Antonio	Get the database on the server
Green	Leo	Prep Servers for hosting

At this point we identified that we would need an Express.Js server to handle the API calls necessary for styling and communication to the database. We successfully migrated both the website and the database to their respective vsphere instances and established communication. We continued to work on the API to build the additional necessary communication between the instances. Work on the Web Server continued.

10/18 Week 8

Yellow	Zack/Kai	Revamping UI styling/layout/design and adding additional pages needed
Yellow	Zack/Kai	Implement account creation, library management
Yellow	Antonio	Create a load testing script to test the servers
Green	Brendon	Set up Express.js server in charge of styling
Green	Kai	UI now uploads images to DB server file system

This week marked a large change in the format for the website. It was decided that it would look better if formatted differently which required large amounts of revamping of the HTML and CSS source code. The Express.js server was fully implemented on the database instance to handle HTTP calls from the webserver and also the styling server, but routes to this Express server would have to be constantly added to as development continued. We decided it was ideal to save images uploaded to our platform to the file system of the database server and then just have our database itself store file paths to the locations of the necessary images as an attribute of the mage itself. We also began to work on the user side of the UI like user profile and library management. It was at this point as well that we began carefully load testing the database server, as it would be queried heavily by both the web server and styling server.

10/25 Week 9

Yellow	Zack/Kai/ Antonio	Revamping UI styling/layout/design
Yellow	Zack/Kai	Implement account creation, library management
Yellow	Antonio	Create a load testing script to test the servers
Yellow	Brendon	Deploy style server
Yellow	Leo/Antonio	Writing scripts to load test DB
Yellow	Antonio/ Brendon	Adjust DB schema to facilitate connection between style server and the database

The revamping of out front end continued full throttle, with the addition of user necessities like account creation and library management. We discovered Jmeter as a method of load testing the database server (before this, multi processing was considered as an attempt to adequately flood the server with inputs). The style server instance was ready to take on the Torch machine learning engine that we had settled on.

11/1 Week 10

Yellow	Zack/Kai/ Antonio	Improve UI styling/layout/design
Green	Zack/Kai	Implement profile and user search functionality
Green	Zack/Kai	Connect UI/frontend to database (with APIs)
Green	Zack/Kai	Implement account creation and modification
Green	Leo/Antonio	Update and make necessary changes to DB
Yellow	Zack/Kai	Implement library management
Yellow	Brendon	Connect style server to database
Green	Leo/Antonio	Writing scripts to load test DB

This week was an especially successful one for our group. Although UI additions and improvements continued as we realized they were probably going to for the duration of the project, the necessary additions of user profile creation, modification and searching were fully functional. The Style server was also completely configured and ready to be connected to the database server in order to retrieve images for processing. The database was fully load tested at this point and didn't fail until nearly 10,000 queries were simultaneously attempted. This came as quite a relief as there was now no question that our website could handle 100 concurrent users.

11/8 Week 11

Green	Zack/Kai/ Antonio	Improve UI styling/layout/design
Green	Zack/Kai	Implement library management
Green	Kai	Complete front-end implementation
Green	Kai/Antonio	Report and admin pages
Green	Kai/Zack	Implement video upload
Green	Brendon	Connect style server to database
Yellow	Zack/Kai	Website animations
Yellow	Zack/Kai	Free user implementation
Yello	Zack/Kai	API authorization headers

The website had taken its final form, including complete user functionality and video uploading. There was also significant work done in the admin reporting portion of the project which required development on the style and API server in order to log these statistics in the first place. We continued to improve the website through fine tuning, animating and bug testing.

11/15 Week 12

Green	Kai	Library auto-refresh
Green	Kai	Form and file upload validation
Yellow	Brendon	Fully support video styling
Yellow	Zack/Kai	Website animations
Yellow	Zack/Kai	Free user implementation
Yellow	Zack/Kai/Antonio	Query refactoring
Yellow	Zack/Kai/Antonio	Admin stats page
Yellow	Leo	Project Paper
Yellow	Leo	Google Hosting Setup
Red	Kai	API authorization headers

Additional functionality added to the website such as auto refreshing users' libraries such that they receive their styled photos and more input validation. All of the existing queries needed to be refactored as well to prevent sql injection. The administrative reporting page continued to be fleshed out. However, we did identify a large issue at this point in how the website authorized with our API. Since the API was only supposed to be accessible to paid users, that meant that nothing else could access the API except for paid users. However, the website isn't a "paid

user," but it needs to call those same APIs in order to function properly (for example, in order to display a user's photos, it has to call the API to get those photos. However, the user who is accessing the website is not a paid user, so if the API was restricted just to paid users, they would not be able to view website data). In the end, a workaround was found, which is listed later in the documentation.

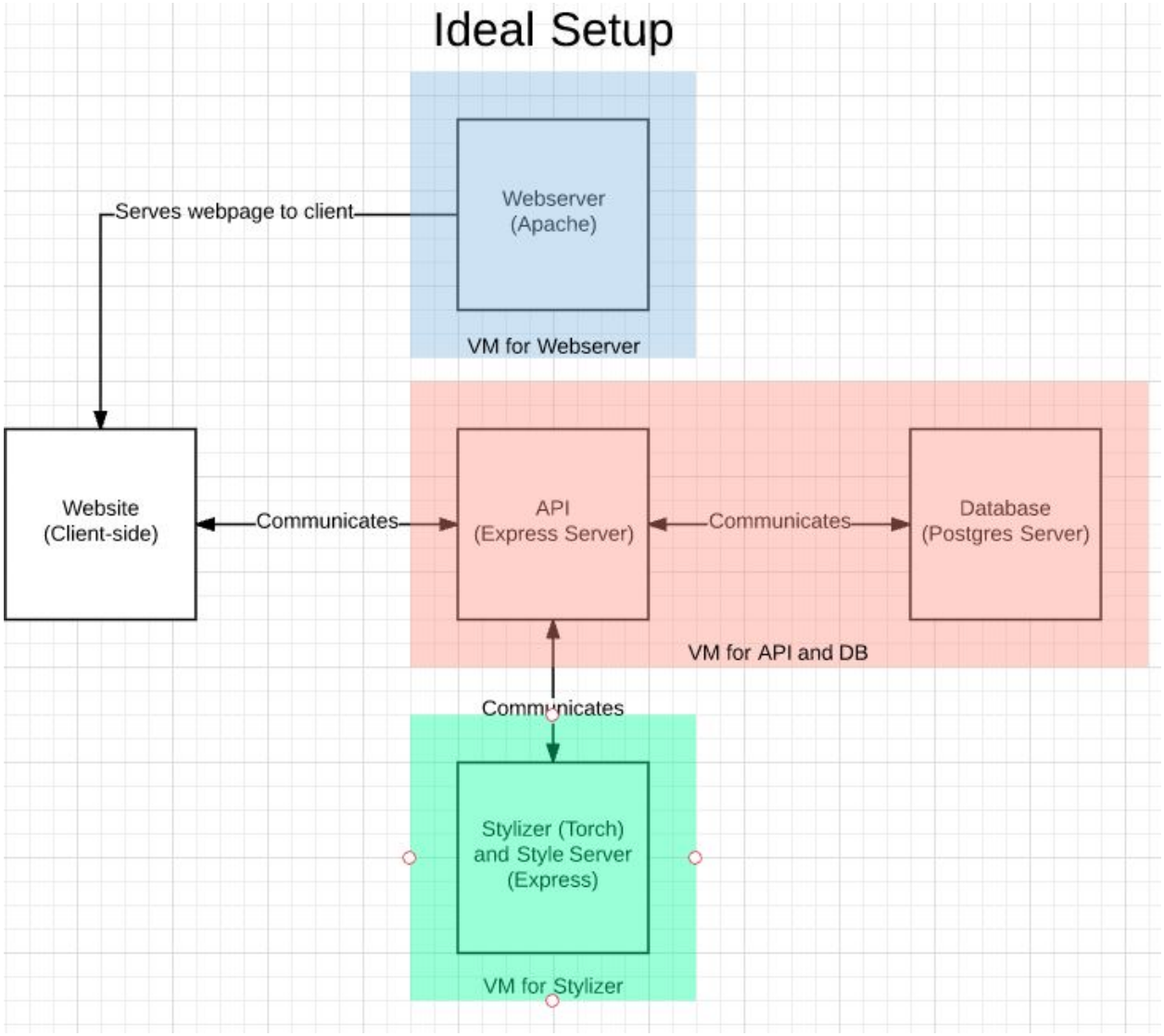
11/29 Week 14

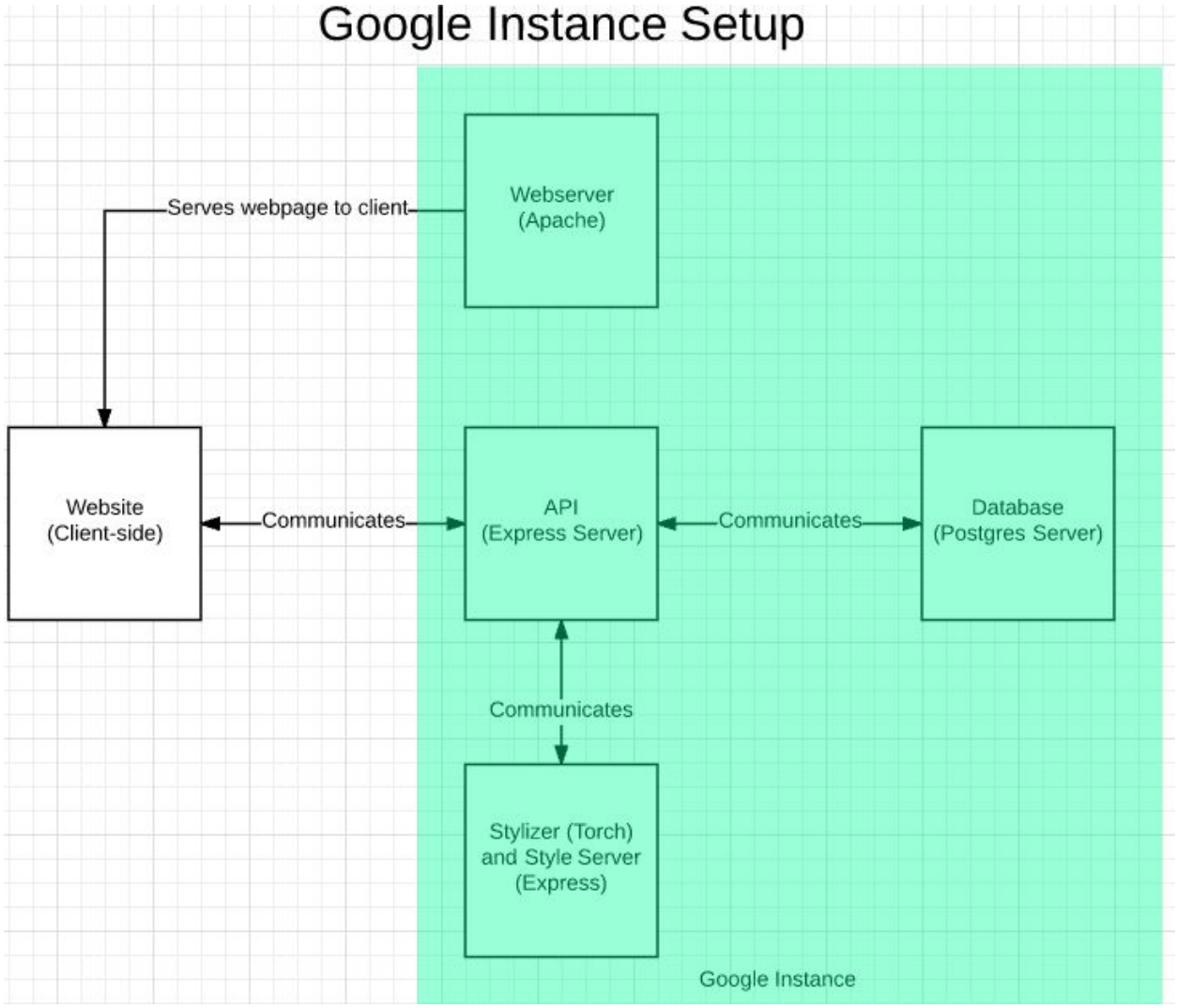
Green	**Antonio**	**Database migrated to production server w/ cleaned data**
Yellow	Brendon	Fully support video styling
Green	Zack/Kai	Free user implementation
Yellow	Zack/Kai/Antonio	Admin stats page
Yellow	Leo/Antonio	Project Paper
Green	Kai	API authorization headers
Yellow	Leo	Help Pages
Yello	Kai	Filesystem deletion
Green	Zack	Stats insertion
Yellow	Zack	Paid user upgrade
Red	Kai	API security

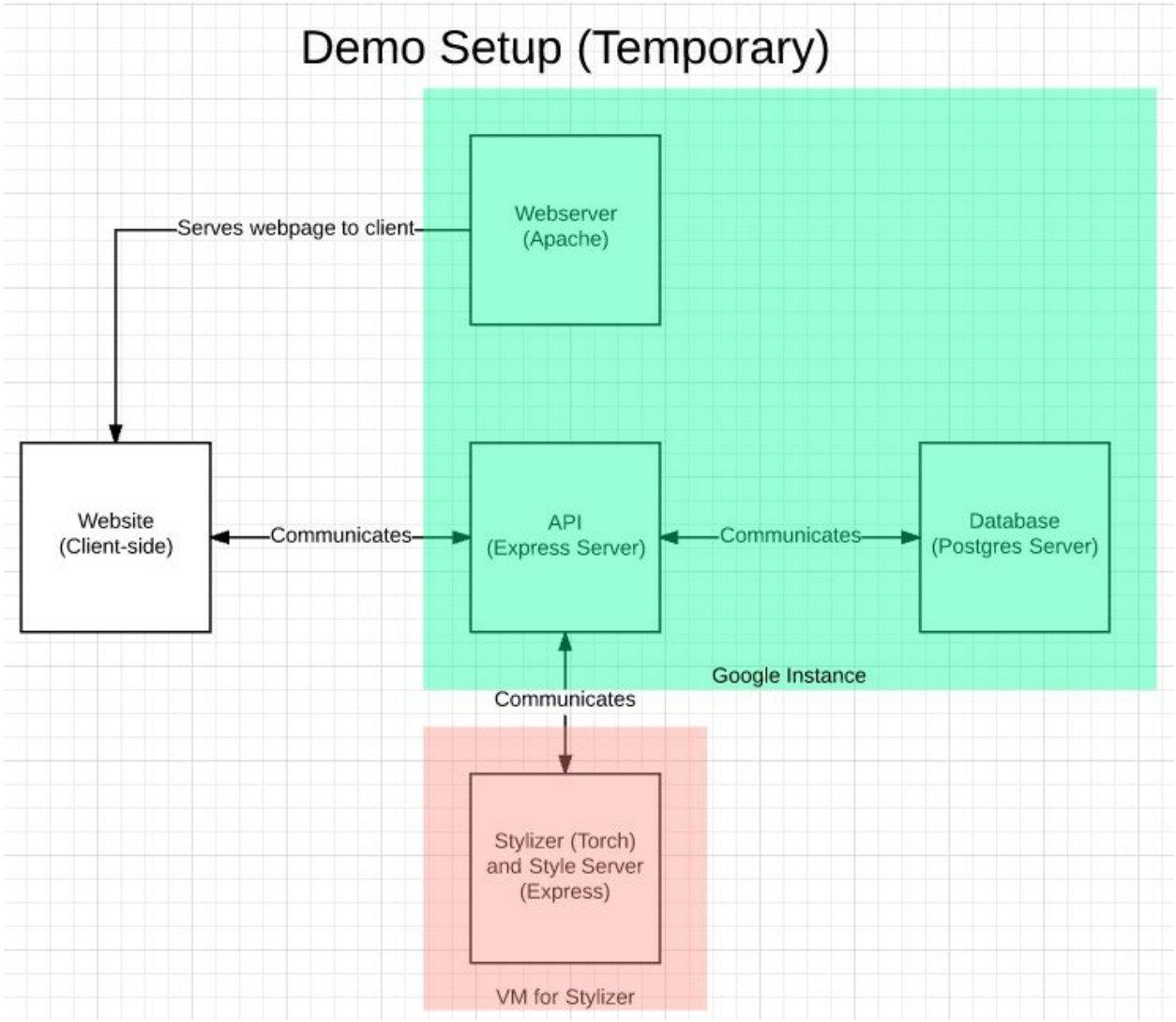
Finally, the last checkpoint between our haul for complete functionality was on 12/6. At this point the stylizer server was experiencing long process times and other problems when styling videos, due to being hosted with limited memory on the Google Instance. The API Authorization problem was resolved, but not without a small caveat. We also needed to fully implement the process of paying for our service and finish documentation such as this document and help pages for our website. We now have strong core functionality.

Architecture

System Layout




Google Instance Setup
Webserver
(Apache)
Serves webpage to client
Website
(Client-side)
Communicates
API
(Express Server)
Communicates
Database
(Postgres Server)
Communicates
Stylizer (Torch)
and Style Server
(Express)
Google Instance



Our ideal setup consists of three VMs/machines:

- One dedicated to hosting the API server and the database server
- One dedicated to running the stylizer and style server on
- One dedicated to running the webserver on to host the web page to users

This allows for an entire machine to be dedicated to the stylizer, with the stylizer VM potentially being equipped with a GPU for faster image processing. A VM dedicated to serving up the database and the API for communication between the client, database, and stylizer, would allow for the database and API server to handle heavy load. Finally, another VM dedicated to serving up the website helps to ensure large amounts of users can responsively load the website.

The website, which is downloaded client-side, communicates with the database and the stylizer through API calls to the Express server.

The Node.js Express server serves these routes, and calling different routes will provide proper CRUD operations on the database to the application or user making those calls. This

Express server is fully equipped with authentication, only allowing trusted clients (the website and the stylizer) to communicate with the API, along with paid users.

The stylizer communicates with the database through the API in order to determine when content is ready to be styled. When styling, the API sends the appropriate content (the image/video and the filter) to the style server, which in turn has the stylizer performs its operations on the content. Upon completion, the style server sends the content back via API, and the database and file system are updated.

For final production mode, we were given only one Google instance on which to host everything. Therefore, we had to put our web server, our stylizer, our API server, and the database all on the same instance. The drawback of this, of course, consisted of our platform being able to handle way fewer customers at a time. This was only made worse with the instance only being equipped with suboptimal technology (4GB RAM, 2.6 GHz CPU single core).

Therefore, for final demo purposes, we temporarily host our stylizer and style server on a separate VM, so that it can have its own dedicated compute power and stylize images in good time. We want to do this in order to demonstrate what our platform is truly capable of. It is very easy to host the stylizer and style server wherever desired; as long as the style server can hit the API, it can served at its location, as all communication between the stylizer and the rest of the system is done through the main API.

Establishing Connection to the Google Instance

Establish Connection to the Server

On Windows

- Generate .ppk file using PuTTYgen, or obtain SSH key from administrator, then launch PuTTY
    - Host Name: developer@imagino.reev.us
    - Leave port default (22)
    - Navigate to Connection > SSH > Auth
        - Browse… for the generated .ppk file (SSH key)
        - Click Open to launch the connection
        - Enter password when prompted

On MacOS/Linux/BSD

- Open terminal
- Navigate to where you saved the private key file
- Connect to the server: **ssh -i imagino.key.txt developer@imagino.reev.us**

Application Design

Webserver

Overview

The webpage is currently being served with Apache.

Front-end Implementation

This is implemented with the latest version of Angular (Angular 4). This is a powerful JavaScript framework that allows for the creation of Single Page Applications (SPA). This allows the website to have extremely fast response times in between loading pages. This works by having Angular route between different components on the page. Learn more at https://angular.io/.

Setup Procedure

1. First, install Apache by running **sudo apt-get install apache2**
2. See later section on how to serve Apache with HTTPS
3. From the user directory, run **git clone https://github.com/brendon-boldt/imago-imaginis**
4. From the web directory in the cloned repo, run **npm install** in order to install project dependencies
5. From the web directory in the cloned repo, run **ng build** to build the web project to the dist folder
6. From the web directory in the cloned repo, run **cp .htaccess dist** in order for Apache to work properly with the Angular webpage
7. Run **sudo a2enmod rewrite**
8. Run **sudo vim /etc/apache2/sites-available/000-default.conf** and change the DocumentRoot to the dist folder (e.g. DocumentRoot originally /var/www/html. Change to /home/developer/imago-imaginis/web/dist)
    a. Also change `AllowOverride None` to `AllowOverride All` under the same DocumentRoot
9. Run **sudo vim /etc/apache2/apache2.conf** and change Directory path to the same path mentioned above (e.g. Directory /home/developer/imago-imaginis/web/dist)
10. Run **chmod -R 755 /home/developer/imago-imaginis/web/dist** to allow Apache to access the web files in dist. Permissions must be set appropriately on the dist folder.
11. Run **sudo service apache2 restart** to restart Apache
12. Visit the website url and you're good to go! (provided all other back-end technologies are running)

Using Certbot to Automatically Install and Renew HTTPS for Apache

- Complete these steps after you have reserved a domain name from GoDaddy or another provider
- Documentation: https://certbot.eff.org/#ubuntuxenial-apache

1. Run the Following commands to install Certbot

   $ sudo apt-get update
   $ sudo apt-get install software-properties-common
   $ sudo add-apt-repository ppa:certbot/certbot
   $ sudo apt-get update
   $ sudo apt-get install python-certbot-apache

2. Run the following command to initiate Certbot service- you will be prompted for the domain of your choosing

   $ sudo certbot --apache

3. To automate the renewal process run the following command

   $sudo certbot renew --dry-run

4. Certbot should now be functioning on your server! You're now running HTTPS for Apache.

User Interface Design

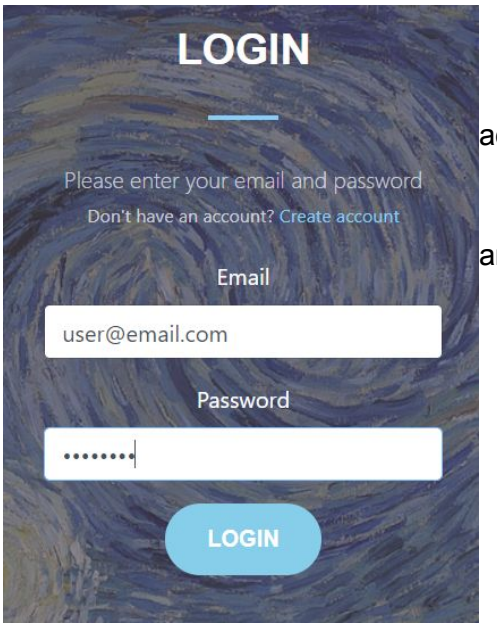
1. Pre Login Home Page

Welcome to the Artistic Stylizer Platform or ASP! We are excited to help you get going on the creative journey our platform provides.

There are several ways to get started it you **do not** have an account yet.

1.2 Click the “Get Started” button at the center of the page (see section 2.1)
1.3 Click the “Log In” button at the top right (see section 2.1)

2. Login Page

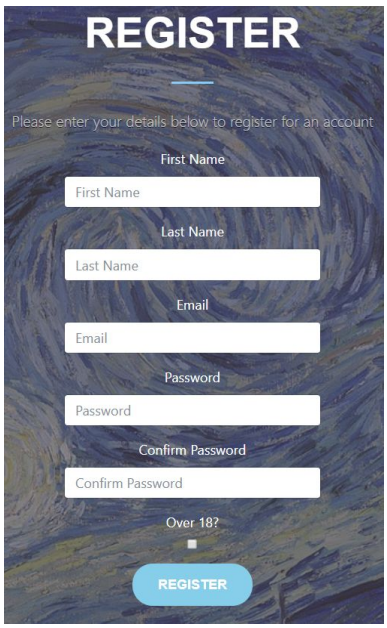


2.1. If you **do not** have an account select the “Create account” button highlighted in blue (proceed to section 3.1)

2.2. If you already have an account enter your username and password and then click the “Login” button at the bottom

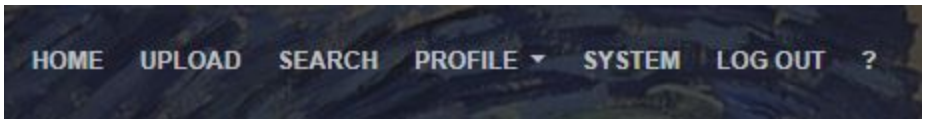
3. Registration Page

3.1. In this page you are invited to create a free account to access our service. This account is necessary to upload photos, stylize them, and create and edit your profile.



3.2. Enter your first and last name, email, and password. You must enter your password twice to confirm they match. Then select the “Over 18?” checkbox (if you are over 18) and you are ready to click the “Register” button at the bottom to create your very own account!

4. Post Login Menu Bar



4.1. This is the menu bar at the top right of website after you log in as an administrator. The only difference between the administrator and regular/paid user menu bar is the regular/paid user does not have access to the “System” button which directs the user to a page displaying system statistics.

4.2. The “Upload” Button directs you to the page where you can upload photos to be submitted to the styling queue. (See section 5.1)

4.3. The “Search” Button directs you to a page where you can search for other user profiles (See section 7)

4.4. The “Profile” button which allows you to view your profile, your image library, or to change your profile information (See section 6)

4.5. The “System” button allows you to view the Administrative Statistics (See section 8) this button is only visible if you are logged in as an administrator

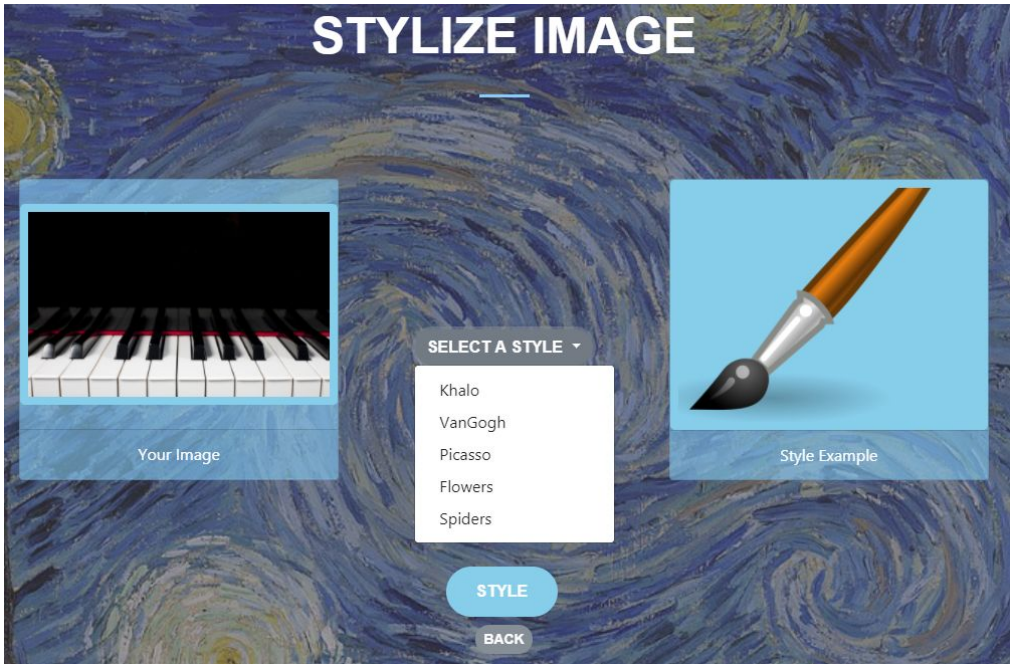
4.6. The “Log Out” button logs you out of your current account

4.7 The “?” button redirects you to this user documentation.

5. Uploading and Styling Images

5.1 This is the upload page where content images may be uploaded. Videos may also be uploaded if they user is a paid user. The first step is to click the “Browse” button in order to select an image from your local device.

5.2 After you have browsed and selected a picture you will enter the style selection screen



5.2.1. In order to select a style, click the “Select A Style” drop down. You will be able to select the one of the default styles with the option to “Upload a Style” as a paid user, in which a content image may be uploaded to be used as the style in the style transfer.

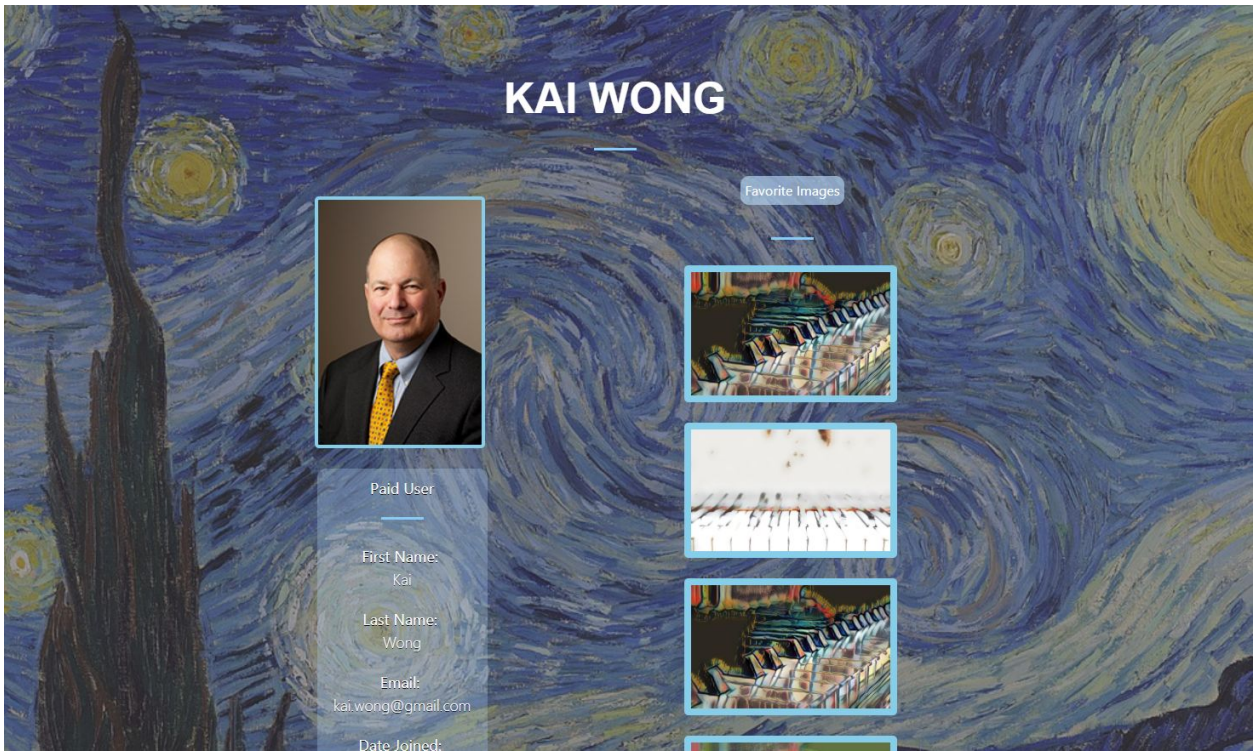
5.2.2. When you are ready to style your image you can select the “Style” button below the selection area to submit the image combination to the styling queue. You may also press the ”Back” button in order to navigate back to the upload screen. You are then redirected to your library page (see section 6.3)

5.2.3 Note that free users’ photos will have their image unobtrusively watermarked.

6. Profile

6.1. When you click on the profile button in the navigation bar, you are given three options in a dropdown: “View” to see your profile (Section 6.2) “Library” to view your photo library (Section 6.3) and “Settings” to edit the information about your profile (Section 6.4)

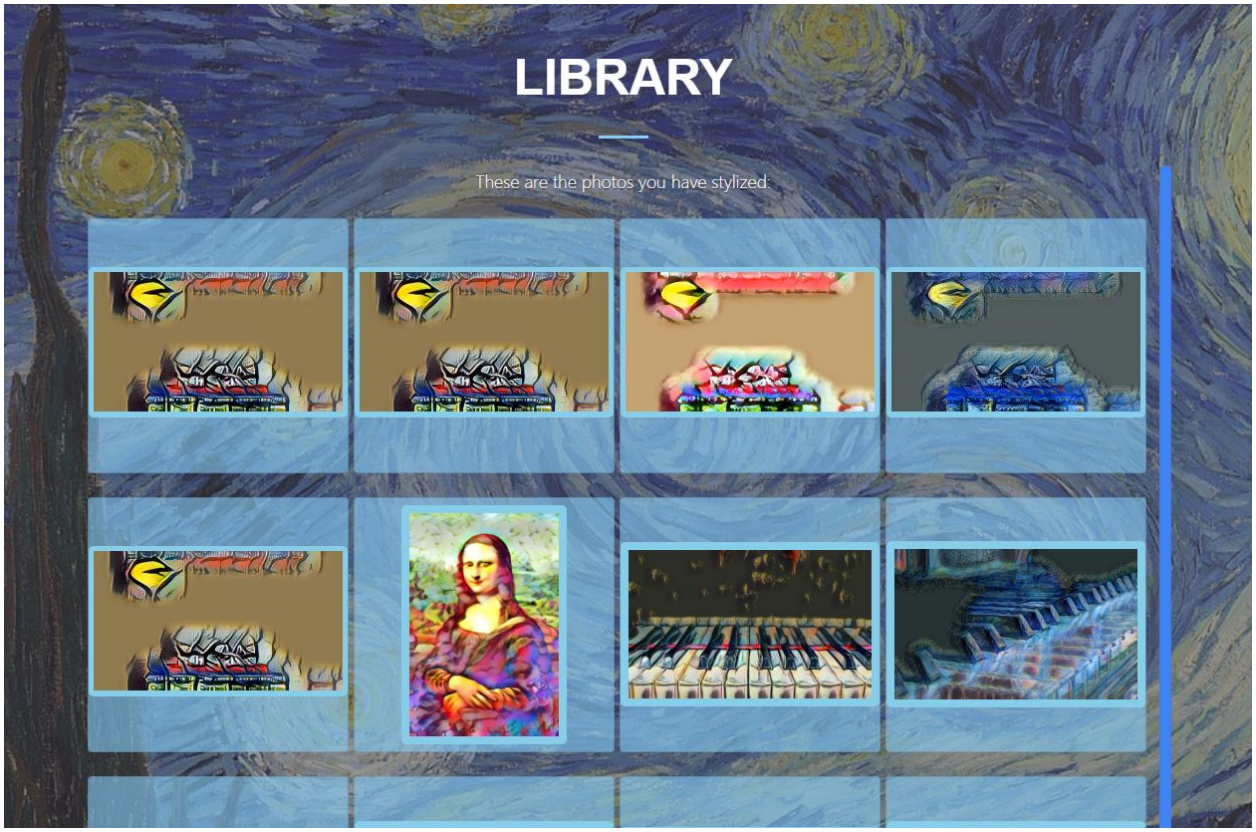
6.2 Profile Page



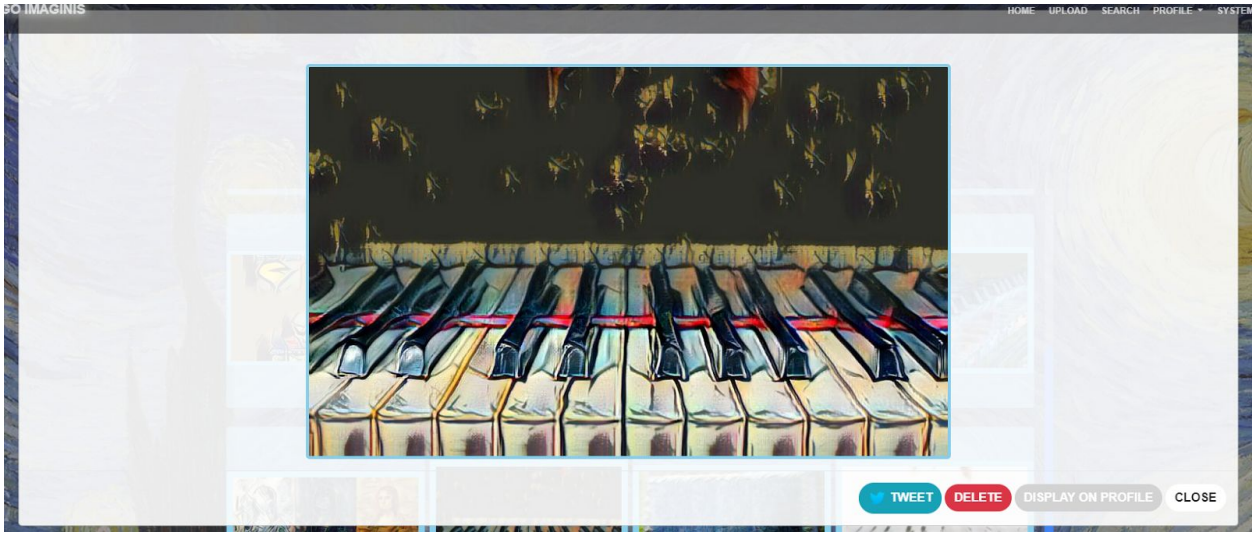
6.2.1 On the left of the profile page, your profile image is displayed, along with an indicator if you are a paid user, your first and last name, your email, and the date you created your profile. On the right, images you choose to display on your profile appear. Your name is displayed prominently at the top of the page. You may click on the photos in order to enlarge them. See section 6.3 and 6.4 for more information on how to update displayed photos and user account information.

6.2.2 Note that free users may only display up to two photos on their profile.

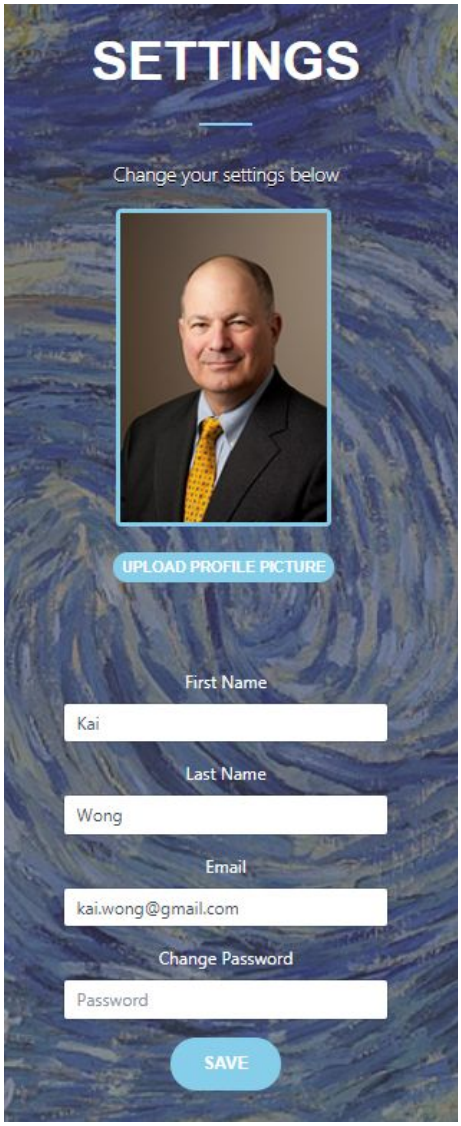
6.3 Library



6.3.1 This is the Library page where paid users can see all their beautiful creations. You may notice some of your images are labeled as “Processing” with a spinning logo. This means that the styling service is still working on them.



6.3.2 Clicking on an image will bring up a dialog in which the full size version of the image is displayed. Here, you may click “Tweet” in order to share your creation to Twitter. You may also to choose to delete the image by clicking the “Delete” button, and you may choose to display or remove the photo on your profile by clicking the “Display on Profile” button. The button will be green if the photo is being displayed on your profile; otherwise, the button is gray.



6.4 User Settings

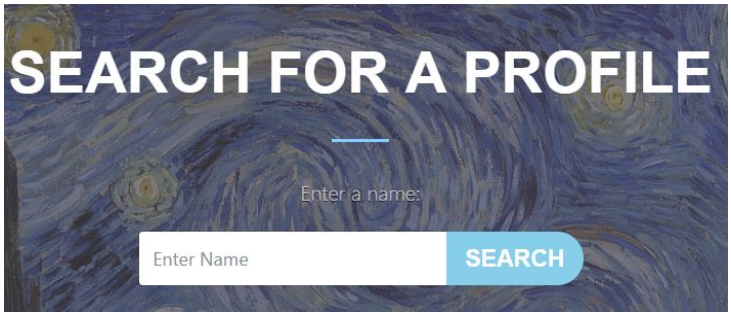
6.4.1 You may select the “Settings” option which brings you to the following screen.

6.4.2 On this screen, you may change any of your information listed in the input fields. You may enter a new first name, last name, email, or password in their respective fields, then click the “Save” Button to commit these changes.

6.4.3 This page also allows you to change your account type from a free to a paid user by selecting the green “Upgrade Account” Button, which brings up a prompt where you can enter credit card information. Once the payment is processed, the account is upgraded to a paid status.

6.4.4 You may also upload a new profile picture in this page by selecting the “Upload Profile Picture” dialog box under your profile picture image box. Selecting this button will open a file explorer so you can upload an image from your local filesystem.

7. Search

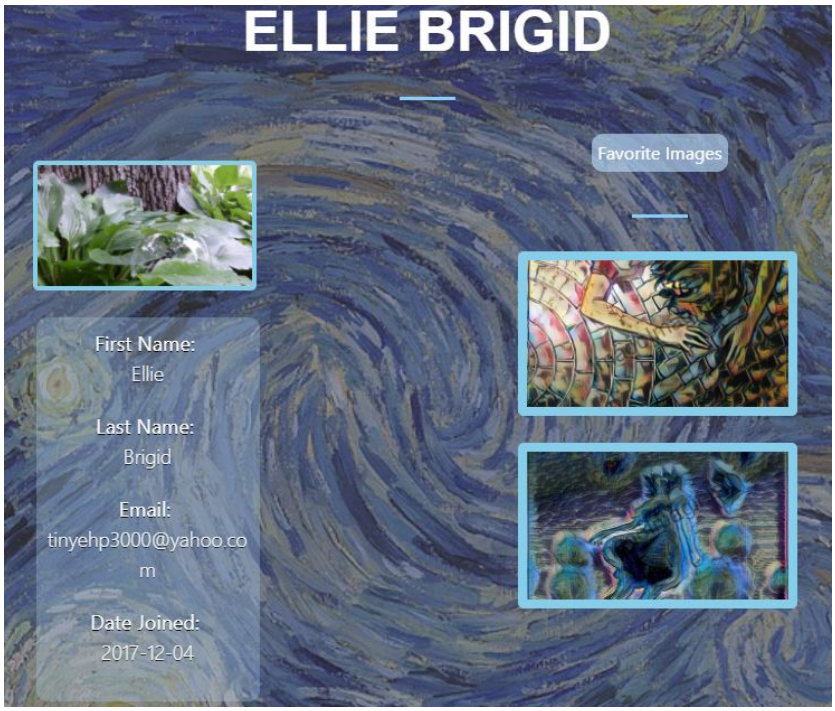


7.1 In this page you are invited to search for another user by any combination of that user's first and last name.

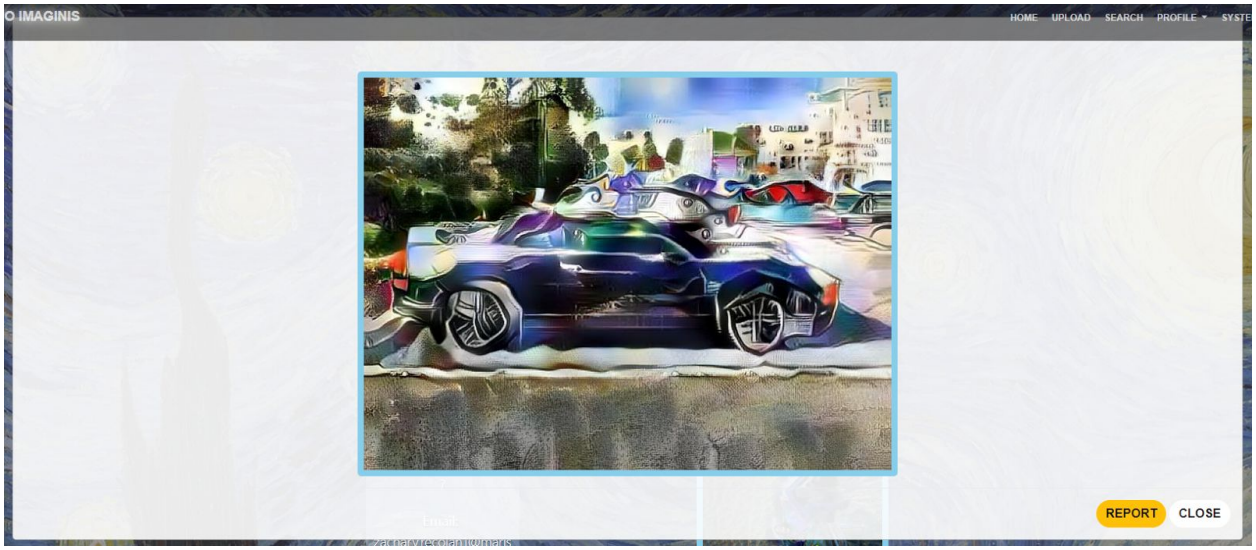
7.2 Note that this page is accessible to the public. This allows anyone to search and view profiles on this platform.

7.3 Enter your search query and press "Search." Results will then be displayed in a table below. Clicking on a result will direct you to the clicked user's profile. See section 8 for more information.

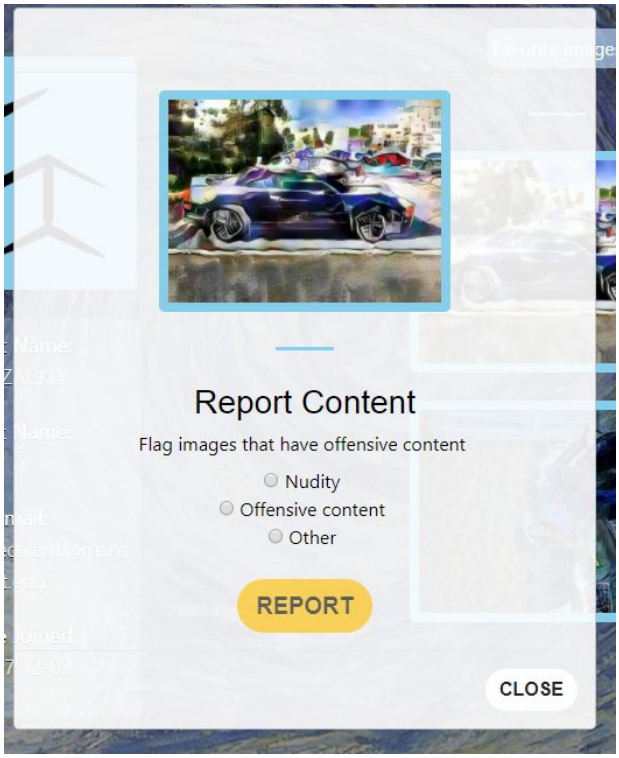
8. Viewing Other Profiles



8.1 Viewing other user profiles is accessible to the public.
8.2 Viewing other user profiles has the same design and functionality as your own profile.

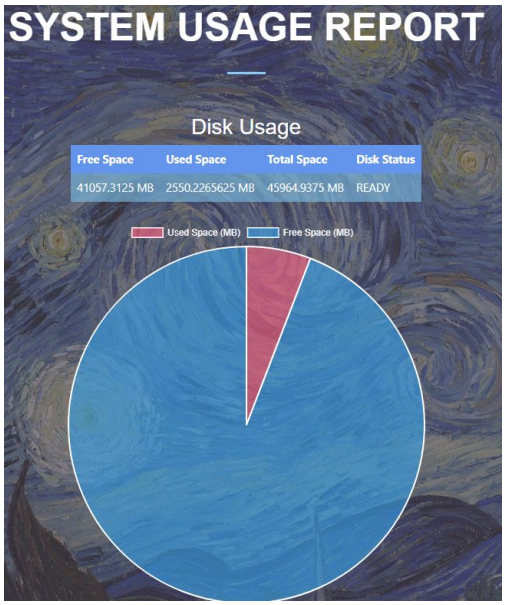


8.3 The modal that is displayed when clicking on a photo to enlarge it now includes a report button. This lets users report content that others may deem inappropriate for the platform.



8.4 Clicking the report button brings up a new dialog which allows you to report an image for a certain offense. This will flag the image in the database, alerting administrators to the offending image.

9. Administrative System Report Page



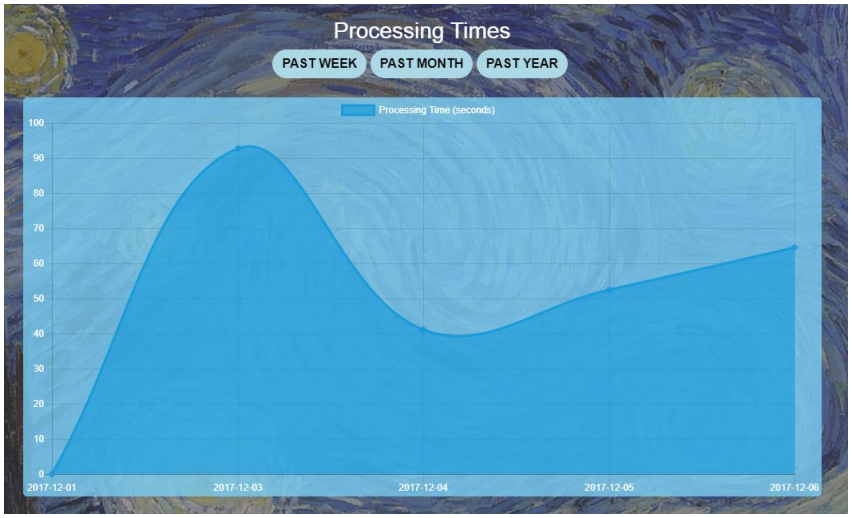
9.1. Selecting the system navbar option when logged in as an **administrator** will redirect you to a page where you can scroll through the system's statistics and usage reports.

9.2 This page displays the following stats:

9.2.1 Disk Usage, which shows free, used, and total space in megabytes on the server, as well as the status of the disk.

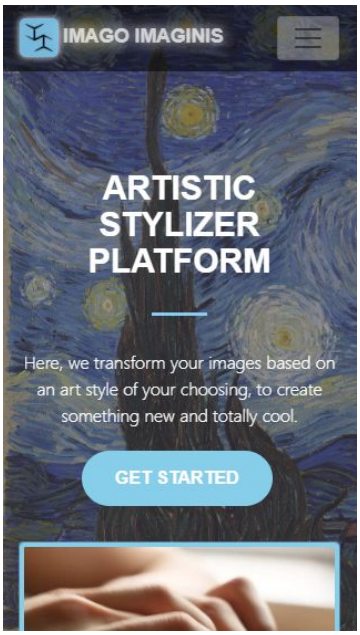
9.2.2 Processing Photos and Videos, which shows the IDs of photos and videos are currently processing, and the filter and user IDs associated with those photos/videos.

9.2.3 Processing Times, which displays a chart of the amount of processing time the stylizer performed within the past week, month, and year. Clicking the buttons at the top of the chart will change the graph to display the appropriate historical information. The scale for processing time is in seconds.



9.2.4 Past Uploads Count and Past Requests Count display a chart of the amount of uploads and requests within the past day, week, and month. Clicking the buttons at the top of the chart will change the graph to display the appropriate historical information.

9.2.5 Reported Photos and Reported Videos display tables with photos and videos that have been flagged as inappropriate content by users. It contains the ID of the photo/video, the ID of the owner of that content, and the date the content was created.



9. Mobile Scaling and Functionality

Our application fully supports mobile scaling and usage for users on small devices such as phones and tablets.

API Server

Overview

This API server, implemented with the Node.js Express package, serves up the routes that act as communication between the different components in the platform (e.g. communication between the client and the database, and between the stylizer and the database). The routes served by the API are mainly called upon by the website; all of the routes are made available to users through the *website*. However, outside of the website, these routes are not accessible; they are only accessible to paid users, who may call upon the API routes for their own purposes

The way the website can access the API is through a special hash that is stored in a hidden way across multiple pages in the website. The API allows access to requests that contain this special hash, in a header called “bus.” If a user were to find this hash scattered throughout the website code, they would be able to construct the hash and access the API as the “website.”

Setup Procedure

Platform assumptions: Ubuntu
Prerequisites software: Node v9+, Git (sudo apt-get install packages)

1. Run **git clone https://github.com/brendon-boldt/imago-imaginis**
2. From the api directory, run **npm install**
3. Setup **config.js** by first running **cp config.js.template config.js** (descriptions of specific fields are contained in the file)
4. Run **node server.js** to start the API (or to detach it from the shell, use **node server.js &** or **tmux** to start a detached session)

Configuring HTTPS on the Express API server

This is assuming that LetsEncrypt is already installed on the Ubuntu instance. Please refer to earlier Apache HTTPS documentation on how to set up LetsEncrypt.

1) Give primkey.pem and fullchain.pem 777 (read, write, execute for everyone) permissions. Both privkey.pem and fullchain.pem are generated by certbot (configured above)
   Example:
   sudo chmod 777 /etc/letsencrypt/archive/imagino.reev.us/privkey1.pem
   sudo chmod 777 /etc/letsencrypt/archive/imagino.reev.us/fullchain1.pem

2) Create hardlinks of privkey.pem and fullchain.pem in /imago-imaginis/api/
   Example:
   sudo ln /etc/letsencrypt/archive/imagino.reev.us/privkey1.pem privkey.pem

sudo ln /etc/letsencrypt/archive/imagino.reev.us/fullchain1.pem fullchain.pem

User Documentation

1. Each route in userRoutes.js, uploadRoutes.js, styleRoutes.js, adminRoutes.js, and webRoutes.js is documented thoroughly in the code. Please see code to see what each routes does and how each route is implemented.
2. Routes were tested through the Restlet Client (https://restlet.com/modules/client/)
3. Routes made available for public paid use. Below are instructions on how to call routes:
    a. /user/login
        i. GET (headers sent with `application/json`)
        ii. Information to pass:
            1. email: email of user
            2. password: password of user
        iii. Information returned: id of user and a new JSON web token to be used to authenticate other API requests
    b. /user/alter
        i. POST (headers sent with `application/x-www-form-urlencoded`)
        ii. Information to pass:
            1. first_name: first name of user
            2. last_name: last name of user
            3. email: email of user
            4. password: (optional) password of user
            5. jwt: the JSON web token to authenticate the user request
    c. /user/search
        i. GET (headers sent with `application/json`)
        ii. Information to pass:
            1. searchString: the search query
            2. jwt: the JSON web token to authenticate the user request
        iii. Information returned: rows of users that matched search query
    d. /user/info
        i. GET (headers sent with `application/json`)
        ii. Information to pass:
            1. user_id: the id of the user to get information from
            2. jwt: the JSON web token to authenticate the user request
        iii. Information returned: all information concerning the user, including if they are a paid user
    e. /user/photos/unstyled
        i. GET (headers sent with `application/json`)
        ii. Information to pass:
            1. user_id: the id of the user to get information from
            2. jwt: the JSON web token to authenticate the user request
        iii. Information returned: all information on the user's unstyled photos
    f. /user/videos/unstyled

i. GET (headers sent with `application/json`)
ii. Information to pass:
    1. user_id: the id of the user to get information from
    2. jwt: the JSON web token to authenticate the user request
iii. Information returned: all information on the user's unstyled videos

g. /user/photos/set-display
    i. POST (headers sent with `application/x-www-form-urlencoded`)
    ii. Information to be passed:
        1. photo_id: id of photo to be displayed on profile
        2. display: true or false depending on if want photo to be displayed
        3. jwt: JSON web token user received from login
h. /user/videos/set-display
    i. POST (headers sent with `application/x-www-form-urlencoded`)
    ii. Information to be passed:
        1. video_id: id of video to be displayed on profile
        2. display: true or false depending on if want video to be displayed
        3. jwt: JSON web token user received from login
i. /user/videos
    i. GET (headers sent with `application/json`)
    ii. Information to pass:
        1. user_id: the id of the user to get information from
        2. jwt: the JSON web token to authenticate the user request
    iii. Information returned: all information on the user's styled videos
j. /user/photos
    i. GET (headers sent with `application/json`)
    ii. Information to pass:
        1. user_id: the id of the user to get information from
        2. jwt: the JSON web token to authenticate the user request
    iii. Information returned: all information on the user's styled photos
k. /user/photos/delete
    i. POST (headers sent with `application/x-www-form-urlencoded`)
    ii. Information to be passed:
        1. photo_id: id of photo to be deleted
        2. jwt: JSON web token user received from login
l. /user/videos/delete
    i. POST (headers sent with `application/x-www-form-urlencoded`)
    ii. Information to be passed:
        1. photo_id: id of video to be deleted
        2. jwt: JSON web token user received from login
m. /user/photos/display
    i. GET (headers sent with `application/json`)
    ii. Information to pass:
        1. user_id: the id of the user to get information from

        2. jwt: the JSON web token to authenticate the user request
    iii. Information returned: information of the photos the user displays on their profile

n. /user/videos/display
    i. GET (headers sent with `application/json`)
    ii. Information to pass:
        1. user_id: the id of the user to get information from
        2. jwt: the JSON web token to authenticate the user request
    iii. Information returned: information of the videos the user displays on their profile

o. /user/photos/num
    i. GET (headers sent with `application/json`)
    ii. Information to pass:
        1. user_id: the id of the user to get information from
        2. jwt: the JSON web token to authenticate the user request
    iii. Information returned: the number of photos the user has uploaded to the platform

p. /filters
    i. GET (headers sent with `application/json`)
    ii. Information to pass:
        1. jwt: the JSON web token to authenticate the user request
    iii. Information returned: information about all preset filters

q. /filter
    i. GET (headers sent with `application/json`)
    ii. Information to pass:
        1. id: the ID of the filter to be looked up
        2. jwt: the JSON web token to authenticate the user request
    iii. Information returned: the path of the filter

r. /report/photo
    i. POST (headers sent with `application/x-www-form-urlencoded`)
    ii. Information to be passed:
        1. photo_id: id of photo to be reported
        2. jwt: JSON web token user received from login

s. /report/video
    i. POST (headers sent with `application/x-www-form-urlencoded`)
    ii. Information to be passed:
        1. video_id: id of video to be reported
        2. jwt: JSON web token user received from login

t. /upload/photo
    i. POST (headers sent with `multipart/form-data`)
    ii. Information to be passed:
        1. upload: photo to be uploaded
        2. jwt: JSON web token user received from login

3. filter_id: id of the filter for the photo to be styled with

u. /upload/video
   i. POST (headers sent with `multipart/form-data`)
   ii. Information to be passed:
      1. upload: video to be uploaded
      2. jwt: JSON web token user received from login
      3. filter_id: id of the filter for the video to be styled with
v. /filter/upload
   i. POST (headers sent with `multipart/form-data`)
   ii. Information to be passed:
      1. upload: photo to be uploaded
      2. jwt: JSON web token user received from login
   iii. Information returned: the filter id of the filter uploaded
w. /user/upload/profile
   i. POST (headers sent with `multipart/form-data`)
   ii. Information to be passed:
      1. upload: photo to be uploaded
      2. jwt: JSON web token user received from login

4. Routes are also available for admins to call
   a. These are all GET (headers sent with `application/json`)
   b. Information to be passed:
      i. jwt: JSON web token user received from login
   c. Routes:
      i. /system/stats (Returns entire stats tables)
      ii. /system/stats/photos/flagged (Returns all flagged photos)
      iii. /system/stats/videos/flagged (Returns all flagged videos)
      iv. /system/stats/photos/processing (Returns all processing photos)
      v. /system/stats/videos/processing (Returns all processing videos)
      vi. /system/stats/uploads/pastday (Returns count of uploads from past day)
      vii. /system/stats/uploads/pastweek (Returns count of uploads from past week)
      viii. /system/stats/uploads/pastmonth (Returns count of uploads from past month)
      ix. /system/stats/reqs/pastday (Returns count of requests from past day)
      x. /system/stats/reqs/pastweek (Returns count of requests from past week)
      xi. /system/stats/reqs/pastmonth (Returns count of requests from past month)
      xii. /system/stats/process/pastyear (Returns total processing time from past year for each day)
      xiii. /system/stats/process/pastweek (Returns total processing time from past week for each day)
      xiv. /system/stats/process/pastmonth (Returns total processing time over past month for each day)

xv. /system/stats/db/spaceused (Returns the amount of space being used by the database)
xvi. /system/stats/filesystem/spaceused (Returns the amount of filesystem space)

Stylizer and Style Server

Overview

1. Stylizer
    a. The stylizer originally comes from https://github.com/xunhuang1995/AdaIN-style which improves the computational efficiency of previous image styling models
2. Style Server
    a. A node.js server which runs where the stylizer is installed.
    b. Responsible for pulling images and videos from the database, starting the stylizing process, and pushing the styled images back to the database.
    c. The style server can reside on any server with https access to the main DB server since the style server communicates with the DB only through https calls to a DB-side API.
3. Style API
    a. Runs with the other APIs on the DB server (no additional setup required)
    b. Handles receiving data and executing database calls requested by the style server

Setup Procedure

4. Install **torch**
    a. Follow install instructions at: http://torch.ch/docs/getting-started.html
        i. Sometimes it is the case that the torch installation process has bug in it. For example, on the Google Cloud production servers, the **moses** package needed to be manually installed in order for all of torch to install properly. (See this issue: https://github.com/torch/torch7/issues/1045)
        ii. On the Marist VMs, the following commands were needed at different points in time for video styling **sudo rm -rf ~/.cache/luarocks ; luarocks install unsup --local** or **luarocks install --deps-mode=all --local unsup**
    b. Configuring torch is a system-specific process, and it is not always as simple as running a few commands; it is often necessary to troubleshoot the individual installation
5. Install the stylizer (the software that will do the actual stylizing)
    a. First, clone the stylizer repository into the home directory (other another location as needed) using **git clone https://github.com/brendon-boldt/AdaIN-style.git**
    b. Follow install instructions
    c. Inside stylizer directory, run the following test command **th test.lua -content input/content/cornell.jpg -style input/style/woman_with_hat_matisse.jpg**

-contentSize 32 -styleSize 32 -gpu -1 -outputName output.jpg -outputDir . the process has completed successfully if there are no errors and **output.jpg** is a small image

d. It is also necessary to install **ffmpeg** for video styling; this can be done simply with **sudo apt install ffmpeg**, but occasionally it is necessary to install ffmpeg from a different source.
e. Inside stylizer directory, run the following test command for videos **bash styVid.sh input/videos/cutBunny.mp4 input/style/woman_with_hat_matisse.jpg output.mp4 32 32** the process has completed successfully if there are no obvious errors and a small, short video is contained in **output.mp4**

6. Install the main project (includes the style server which communicates with the database)
    a. Run **git clone https://github.com/brendon-boldt/imago-imaginis.git**
        i. If the server is already installed, it is not necessary to clone the repository again (but it should be updated with **git pull**)
    b. Navigate to the style server directory using **https://github.com/brendon-boldt/AdaIN-style**
    c. Install node packages using **npm install**
    d. Set up **config.js** by first running **cp config.js.template config.js** (descriptions of specific fields are contained in the file)
    e. Run **npm start** to start the server (or to detach it from the shell use **npm start &**)

Performance Testing the Stylizer

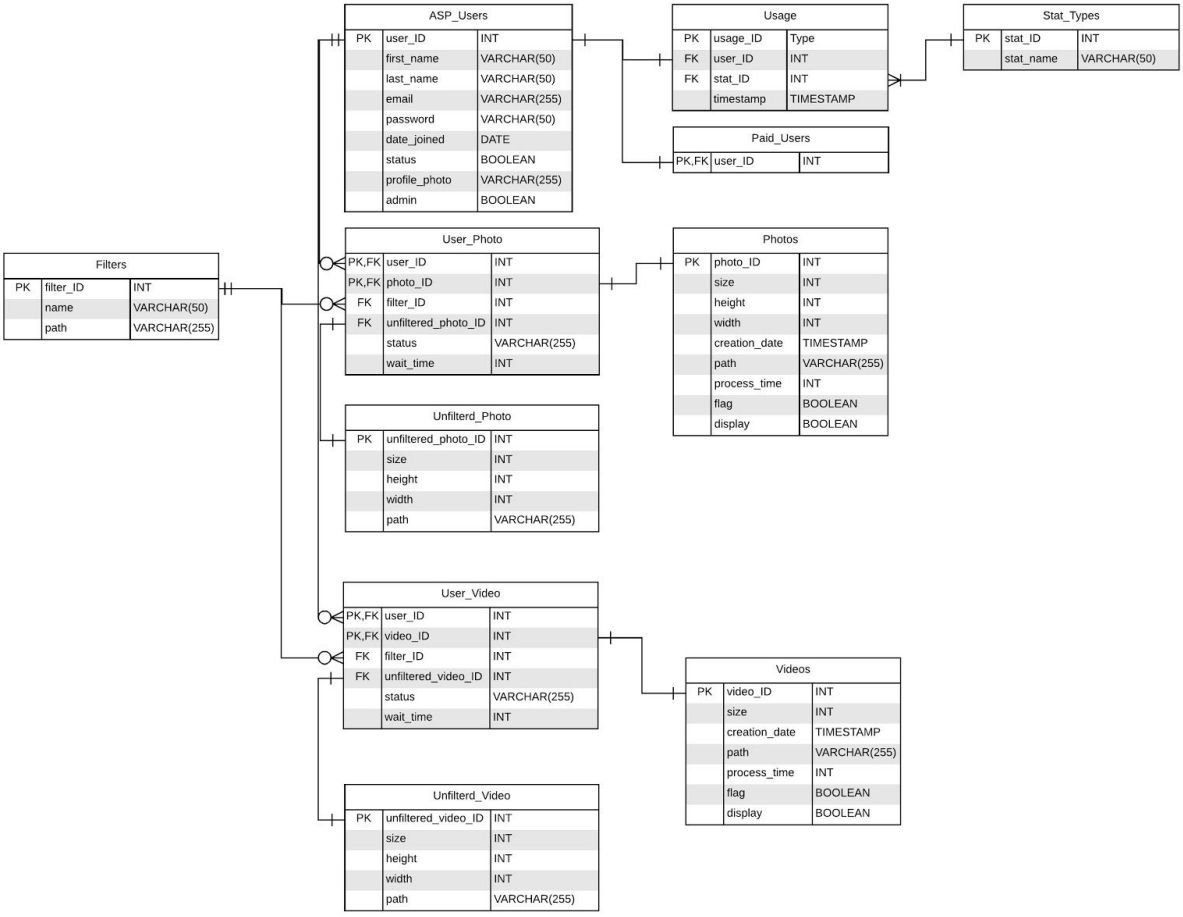
- CPU: Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz (4 cores)
- Memory: 16049 MB
- Beta testing results (time spent in styling)
    - 26 images styled
    - Average: 8068 ms
    - Standard Deviation: 3599 ms
- Tables measure total processing time for each resolution for JPG, PNG, and MP4
    - Images have a fixed output of 512px in the smaller direction (aspect ratio unchanged)
    - Videos have a fixed output of 455x256 (16:9)
    - Styled with a 591x800 image
    - Results show performance little variance based on input for a fixed output size

Image Resolution	JPG Time (s)	PNG Time (s)
720×480	7.023	7.043
1280×720	7.185	7.178
1920×1080	7.250	7.184
4096×2160	7.642	7.620
8192×4320	7.939	7.992

Video Resolution	MP4 FPS
720×480	0.756
1280×720	0.776
1920×1080	0.783
4096×2160	0.774
8192×4320	0.737

Database Server

Design



Database Users

DBA:

Username: administrator
Password: Cg17
Description:

As the DBA, this user will have total control/power over the database, including having the ability to create tables, drop tables, and alter table structures. Provided access to all DDL and DML commands.

Users:

Username: wsadmin
Password: Cg17

Description:

As the wsadmin, this user will have SELECT, INSERT, UPDATE, and DELETE permissions on all tables.

Username: wsuser
Password: Cg17
Description:

As the wsuser, this user will have SELECT, INSERT, UPDATE, and DELETE permissions on all tables.

Design Notes

- Email is stored as a length up to 255 characters because of the SMTP standard
- We store usage/process statistics with the Photo and Video entities, and other statistics in the Usage entity. These other statistics could be a page visit, a site visit, or more. From this, we can derive how many page/site visits there were in a certain time frame. We can also derive how many photos and videos were uploaded from the statistics given by the Photo and Video entities, and we can derive total process time over a given time frame.
- Process time is an integer that represents the number of seconds it took for an image to finish processing
- We won't need to store any parameters for the deep learning component
- All Photos and Videos are saved in their respective tables as paths to directories in the Database Server's File System

Sample Queries for Backend/Admin Users

- Only administrator can access our backend.
- In order to perform usage/statistical queries, one may perform queries such as the following:
  - To get the total processing times within the past month:
    - ```
      SELECT creation_date, SUM(process_time) FROM photos WHERE
      (creation_date BETWEEN now() - INTERVAL '1 MONTH' AND now())
      GROUP BY creation_date ORDER BY creation_date;
      ```
  - To get total uploads within the past week:
    - ```
      SELECT CAST(TIMESTAMP AS DATE), COUNT(*) FROM usage,
      stat_types WHERE stat_types.stat_id = usage.stat_id AND
      (TIMESTAMP BETWEEN now() - INTERVAL '7 days' AND now()) AND
      (usage.stat_id = 2 OR usage.stat_id = 3) GROUP BY
      cast(TIMESTAMP AS DATE) ORDER BY timestamp
      ```
  - To get photos being processed:

        - `SELECT user_photo.unfiltered_photo_ID, user_photo.filter_id, user_photo.wait_time, user_photo.user_id FROM user_photo WHERE status = 'processing'`
    - To get photos that have been reported:
        - `SELECT photos.photo_id, user_photo.user_id, photos.creation_date FROM photos, user_photo WHERE photos.photo_id = user_photo.photo_id AND photos.flag = true`
- Example of more queries can be found located under their respective routes in the various route files in the GitHub repo project under api/app/routes

Database Setup Procedure

Install Postgres

- Update packages: **sudo apt-get update**
- Install Postgres: **sudo apt-get install postgresql**
- Switch to the Postgres user that was just created: **sudo -i -u postgres**
- Launch Postgres: **psql**

Configure Postgres

- Update **pg_hba.conf** to allow mv5 password login for all local users
    - After launching psql: **SHOW hba_file;**
        - Note the path
    - Quit psql: **\q**
    - Open pg_hba.conf: **sudo vim /path/to/pg_hba.conf**
    - Alter the line: `local all all trust` to `local all all mv5`

- Update **pg_hba.conf** to allow mv5 login over ssh for all ips
    - Add line: `host all all 0.0.0.0/0 mv5`
    - Save and quit vim
    - Restart Postgres: **sudo service postgresql restart**

- Update **postgresql.conf** to listen for all incoming traffic
    - Open postgresql.conf: **sudo vim /path/to/postgresql.conf**
    - Alter the line: `# listen_addresses = 'localhost'` to `listen_addresses = '*'`
    - Save and quit vim
    - Restart Postgres: **sudo service postgresql restart**

Create The Users

- Launch Postrgres if not open
- administrator: **CREATE USER administrator;**
    - should be a superuser

- wsadmin: **CREATE USER wsadmin;**
    - should **NOT** be a superuser
- wsuser: **CREATE USER wsuser;**
    - should **NOT** be a superuser
- Create passwords: **ALTER USER <user> with password <password>;**
    - Do for each user

Grant Permissions to Users

- Launch Postrgres if not open
- Grant required permissions for each user: **GRANT <priv> ON <table> TO <user>;**
    - Note: roles are defined in pg_dumps. If you are restoring a database from a backup, you simply need to create the roles, and then the permissions will be granted upon restoring the database.

Create the Database

- Launch Postgres if not open
- Create the database: **CREATE DATABASE aspdb;**

Load a Database Backup / Initial Data onto the Server

- Quit Postgres if open
- Load the backup file: **:$ cat /path/to/backup.sql | psql aspdb**

The database should now be completely up and running.

Accessing the ASPDB Database, From Start to Finish

- SSH into the server from the OS of your choice
- Switch to the Postgres user: **sudo -i -u postgres**
- Connect to the DB with appropriate credentials:
    - **psql aspdb wsadmin**
    - **psql aspdb wsuser**
    - **psql aspdb administrator**
- Enter the appropriate password, and you're in.

Creating a Database Backup

- Navigate to the directory where you want to save a backup
- Dump the database to the directory: **pg_dump aspdb -U administrator > backup.sql**
    - Backup.sql can be renamed to any descriptive filename, usually with a date
- Enter your password, and it will be saved to that directory

Restoring a Database Backup

- See '**Load a Database Backup / Initial Data onto the Server'** above

Load Testing the Server with ApacheBench

- Ensure that Apache is installed on your local machine. If it is not, install Apache. The installation will also install ApacheBench, the load testing software that we will be using.
- Open the terminal and navigate to the directory of an image file that we will test posts with.
- execute the following command:
    - **ab -r -n <# hosts> -c <# concurrent requests> -p <image file> <server address>**
        - -r: don't exit on socket receive errors
        - -n: number of requests to perform on the benchmarking session
        - -c: number of concurrent requests to perform
        - -p: file containing data to POST

Infrastructure Design

IT Requirements

Database Server

1.1. Physical system requirements
    1.1.1. Storage Capacity - 15Mb for each free user, 50Gb for paid (5-10tb initial storage at launch). Since the max image size a free user can upload is 7Mb, and they can only store two stylized pictures, we give them 15Mb (1Mb for extra leeway). We give 50Gb for paid users, which we think gives them enough space for all their pictures/videos.
    1.1.2. Speed requirements / response time parameters - High response time, because we want response times to be instant, so that the platform's performance does not suffer from lengthy database access times.
    1.1.3. Scalability plans - Easily scalable storage capacity possibly through use of a storage area network. This is for preparation for the future.
1.2. Virtual system requirements
    1.2.1. OS to be supported - Ubuntu 16.04
    1.2.2. Number of images expected - 2; 1 Primary 1 Backup. We will keep a backup in case the primary becomes corrupted.
1.3. Connectivity
    1.3.1. Network considerations - N/A
    1.3.2. Interconnection to what other systems - Connected to stylizer server and web server

Stylizer Server

1.1. Physical system requirements

1.1.1. Storage capacity - 75GB. Since photos are stored in the database, the only photos that are stored on the stylizer server are the photos that are in the job queue waiting to be stylized. After they are stylized, they are sent to the database server and removed from this server.
1.1.2. Speed requirements / response time parameters - High processing speeds, multiple hyper threaded processors and a dedicated GPU. This is so that the stylizer is given the compute power needed in order to maximize its performance, thus giving decreasing time user has to wait for their image processing.
1.1.3. Memory requirements - Installing torch requires approximately 3gb of memory. Since the photo stylization process is very memory intensive, 3.5gb of memory will yield output photos with a resolution of approximately 400x300 (and the server would only be able to process one job at a time).
1.1.4. Scalability plans - None, it just has to have available space for patches to the ASP, and for the temporary photos held in the job queue.

1.2. Virtual system requirements
1.2.1. OS to be supported - Ubuntu 16.04
1.2.2. Number of images expected - 2; 1 Primary 1 Backup. We will keep a backup in case the primary becomes corrupted.

1.3. Connectivity
1.3.1. Network considerations - N/A
1.3.2. Interconnection to what other systems - Connected to database server

Web Server

1.1. Physical system requirements
1.1.1. Storage capacity - Very Low 20GB. Nothing is uploaded directly to the web server, and the website itself is not very large. We will not store much on the web server.
1.1.2. Speed requirements / response time parameters - High response Time, lots of RAM. We want the website to be responsive and load quickly for the user.
1.1.3. Scalability plans - Should be able to add more RAM, processing power, and networking capabilities, or just more web servers as platform popularity grows
1.2. Virtual system requirements
1.2.1. OS to be supported - Ubuntu 16.04
1.2.2. Number of images expected - 1 initially, with 1 backup. We will keep a backup in case the primary becomes corrupted.
1.3. Connectivity

1.3.1. Network considerations - Several NICs (2 for now) and redundant, high capacity network connectivity. This is to allow high network traffic to pass through without being bottlenecked by the server's network connectivity.
1.3.2. Interconnection to what other systems - Database Server

Reliability

2.1. Service Level Agreements

2.1.1. Uptime requirements - Our Web Server needs to be up 99.99% of the time, we will have scheduled maintenance (weekly basis) on the Stylizer and Database server, and for those periods our service will be unavailable. We want the website to be accessible at all times, and we think that this metric is possible, because of strong web server technology and security. However, 100% uptime is not possible in a real-world situation, so we only guarantee 99.99%.

2.1.2. Response time requirements - Our user should be able to have their picture converted in a matter of seconds. If they upload a video, it may take several minutes depending on the length, but any process running for over an hour will be terminated. The website should have an instant response time so that the user is not discouraged from using the site.

Recoverability

3.1. Where are things backed up? How often? - The ASP will save two copies of each image for our paid users, one primary and one backup, at the time of its creation. There will also be a backup written for all user information at the time of its creation. We will encourage free users to save their created images locally.

3.2. Access to backups? - The paid user will not have access to their backup images unless the primary is compromised, at which point the backup becomes the primary data source. Free users will not have any backups.

3.3. What data is transient and doesn't need to be stored longer term? - Any data stored or cached by the web server will not need to be stored long term. In addition, the base image the user uploads will not need to be stored.

Security and Privacy

4.1. Database

4.1.1. Access controls by user id / roles - Users will login using a user id that determines their membership level. System administrators will log in to their backend reporting separate from the front facing website, and also have a user id for the database access tool.

4.1.2. Update vs. Access - users will be able to access their images stored on the database through the website, as well as view other users' showcased ones.

The only way a user can update the database is by submitting an image to be stylized through the web interface, removing an image from their profile, or updating their account settings. Users will not be able to delete anything from the database. The administrators will have full database access through a separate application to run SQL commands against

4.1.3 Usernames - There will be one admin username, which allows the user to have full database and reporting control. There will be one username for application purposes, which allows the user to have limited database control (no create, delete, or drop, only read and update). Finally, there will be another username for reporting purposes, which just allows the user to query statistical information.

4.2. Account information

4.2.1. User data - Users will store very minimal personal data on our site; their name, email, a profile picture, and a short bio. The upgrade to a premium account will be a one time cost to a credit card, so it will not be stored. Passwords will be encrypted with SHA256.

4.2.1.1. Personal / registration - The registration of a user will be linked to an email account of theirs.

4.2.1.2. Saved information - We will use session variables to store temporary session data

4.3. Admin access controls

4.3.1. Adding new users, deleting old - Admins will be able to send out warnings that free users accounts will be deactivated after inactivity for 2 years. We think 2 years is a good amount of time to detect that an account is not being used anymore. Admins will not be in charge of creating new users, but will have the ability to do so. Paid users keep their accounts permanently.

Maintenance

5.1. Planned down time requirements

5.1.1. Database and stylizer maintenance - maintenance of these services will occur as needed during non peak hours after a notification posted a day ahead on the site. During this period, the database and the stylizer will be down.

5.1.2. Site maintenance - the site can be instantly updated, and will stay up during maintenance.

5.1.3. Times of year when IT does maintenance - Weekly basis during off-peak hours (night-time). This way, it’s a balance of having the least amount of users impacted and the platform staying well-maintained.

5.1.4. Times of year when the systems are not available? - It will be available a vast majority of the time of the year as it is not a seasonal application.

VSphere Configuration for Development

These configuration steps will lead you through the initialization of the VSphere instances used for the application's development. They will change dramatically depending on the user's own infrastructure.

Database Server Instance Configuration

1. Configuration: Custom
2. Name and Location:
    a. Name: <User Preference>
    b. Inventory Location: <User Preference>
3. Storage: <User Dependent>
4. Virtual Machine Version: 8
5. Guest Operating System: Linux
    a. Version: Ubuntu Linux (64-bit)
6. CPUs:
    a. Number of virtual sockets: 1
    b. Number of cores per virtual socket: 2
7. Memory:
    a. Memory Size: 4096 MB
8. Network:
    a. How many NICs do you want to connect?: 1
    b. NIC 1:
        i. Network: VM Network
        ii. Adapter: VMXNET 3
        iii. Connect at Power On: Yes
9. SCSI controller: LSI Logic Parallel
10. Select a Disk: Create a new virtual disk
11. Create a Disk:
    a. Disk Size: 5 TB
    b. Disk Provisioning: Thin Provision
    c. Location: <User Preference>
12. Advanced Options
    a. Virtual Device Node: SCSI(0:0)
    b. Mode: Not Independent

Now you are ready to load Ubuntu Linux 16.04 iso onto your instance

Stylizer Server Instance Configuration

13. Configuration: Custom
14. Name and Location:

    a. Name: <User Preference>
    b. Inventory Location: <User Preference>
15. Storage: <User Dependant>
16. Virtual Machine Version: 8
17. Guest Operating System: Linux
    a. Version: Ubuntu Linux (64-bit)
18. CPUs:
    a. Number of virtual sockets: 1
    b. Number of cores per virtual socket: 4
19. Memory:
    a. Memory Size: 16384 MB
20. Network:
    a. How many NICs do you want to connect?: 1
    b. NIC 1:
        i. Network: VM Network
        ii. Adapter: VMXNET 3
        iii. Connect at Power On: Yes
21. SCSI controller: LSI Logic Parallel
22. Select a Disk: Create a new virtual disk
23. Create a Disk:
    a. Disk Size: 100 GB
    b. Disk Provisioning: Thin Provision
    c. Location: <User Preference>
24. Advanced Options
    a. Virtual Device Node: SCSI(0:0)
    b. Mode: Not Independent

Now you are ready to load Ubuntu Linux 16.04 iso onto your instance

Web Server Instance Configuration

25. Configuration: Custom
26. Name and Location:
    a. Name: <User Preference>
    b. Inventory Location: <User Preference>
27. Storage: <User Dependant>
28. Virtual Machine Version: 8
29. Guest Operating System: Linux
    a. Version: Ubuntu Linux (64-bit)
30. CPUs:
    a. Number of virtual sockets: 1
    b. Number of cores per virtual socket: 2
31. Memory:

    a. Memory Size: 4096 MB
32. Network:
    a. How many NICs do you want to connect?: 1
    b. NIC 1:
        i. Network: VM Network
        ii. Adapter: VMXNET 3
        iii. Connect at Power On: Yes
33. SCSI controller: LSI Logic Parallel
34. Select a Disk: Create a new virtual disk
35. Create a Disk:
    a. Disk Size: 80 GB
    b. Disk Provisioning: Thin Provision
    c. Location: <User Preference>
36. Advanced Options
    a. Virtual Device Node: SCSI(0:0)
    b. Mode: Not Independent

Now you are ready to load Ubuntu Linux 16.04 iso onto your instance

Miscellaneous Documentation

Link to GitHub Repository: https://github.com/brendon-boldt/imago-imaginis
Link to Demo Website: https://imagino.reev.us

Test Cases for Site Input

Login
- User must enter their email and password
- Test cases:
    - Try to enter nonexistent login information

Upload
- User must upload a valid image file (.PNG or .JPG), or a valid video file if they are a paid user (.MP4)
- Test cases:
    - Try to select a file that is not a .PNG or .JPG
    - Try to select a file that is an .MP4 while a non-paid user

Style
- User must upload an image if they choose to upload their own style
- Test cases:
    - Try to upload a filter that is not a .PNG or .JPG

Search
- Test cases:
    - Enter a user that doesn't exist

Profile

- Settings
    - User must upload a valid image file as their profile picture (.PNG or .JPG), and they must fill the form out completely (besides the password, which is optional)
    - Test cases:
        - Try to upload a file that is not a .PNG or .JPG
        - Try to submit an empty field (password is an optional field)
        - Try to submit name fields with only spaces
        - Try to submit an invalid email

Sign Up

- User must insert fill out fields and check the age gate in order to complete this form
- Test cases:
    - Try to submit an empty field
    - Try to submit name fields with only spaces
    - Try to submit passwords that don't match
    - Try to submit an invalid email
    - Try to not click the age gate

Test Cases for API Input

General:

- The API is only accessible to paid users
- Test cases:
    - Try to call any route without a valid JWT (besides /user/login)

Login: (/user/login)

- The API user must provide all fields
- Test cases:
    - Try to call this route without supplying a password
    - Try to call this route with credentials of a free user

Upload: (/upload/photo)

- The API user must provide all fields and supply a valid .PNG or .JPG
- Test cases:
    - Try to call the route with a non-image file
    - Try to call the route without supplying a filter id

Retrieve Styled Photos: (/user/photos)

- The API user must provide the user id of a user in order to see their styled photos
- Test cases:
    - Try to call the route without a user id

Content Deletion: (/user/photos/delete)

- The API user can only delete content that belongs to them
- Test cases:
    - Try to delete a photo that does not belong to the API user

Load Testing Results: Production Server (Google Instance)

As it stands, the server seems capable of handling 200 concurrent users without failures. Tests were performed by sending concurrent POST requests, each containing a jpg image file, to simulate various numbers of concurrent users uploading images to the server. Testing was done using ApacheBench, the operation instructions of which have been documented under the database section of this document. The command that was executed was:

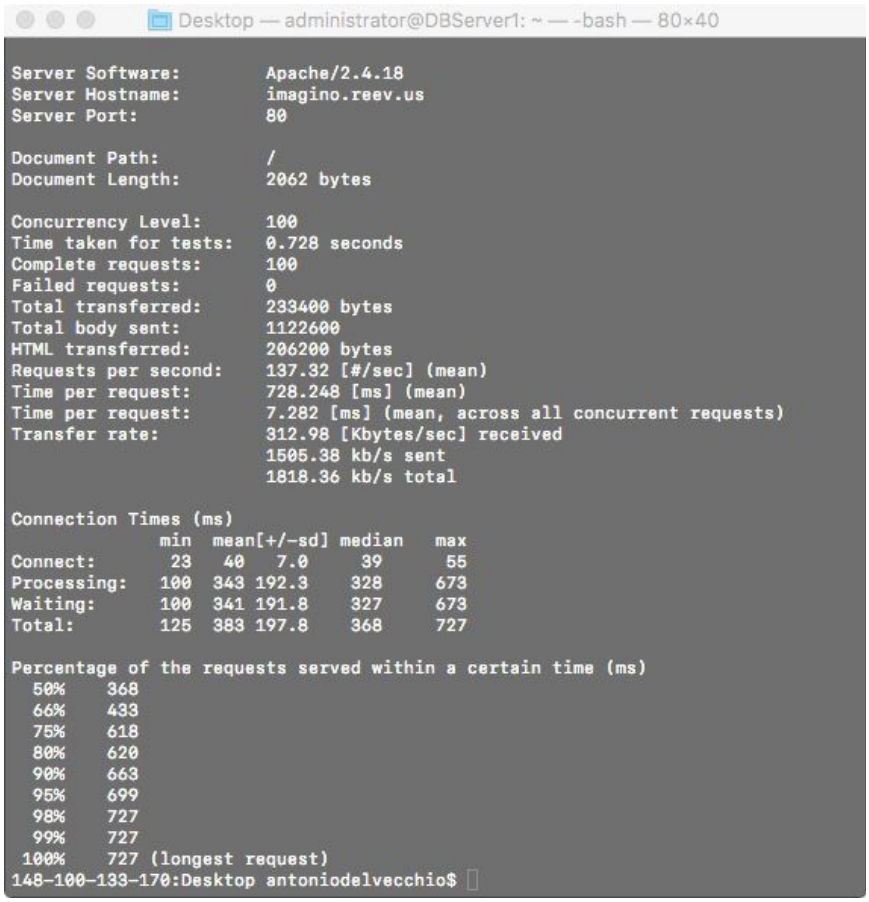
ab -r -n <# hosts> -c <# concurrent requests> -p <image.jpg> https://imagino.reev.us/.

It should be noted that an absence of trailing slashes has been known to cause errors in some cases, so be sure to include them in the server address.

The project requirements called for at least 100 concurrent users, so this was the first threshold that was tested. In this case, the command was:

ab -r -n 100 -c 100 -p 4d3d3.jpg https://imagino.reev.us/

This sends 100 concurrent POST requests to the server. The results were printed as follows:

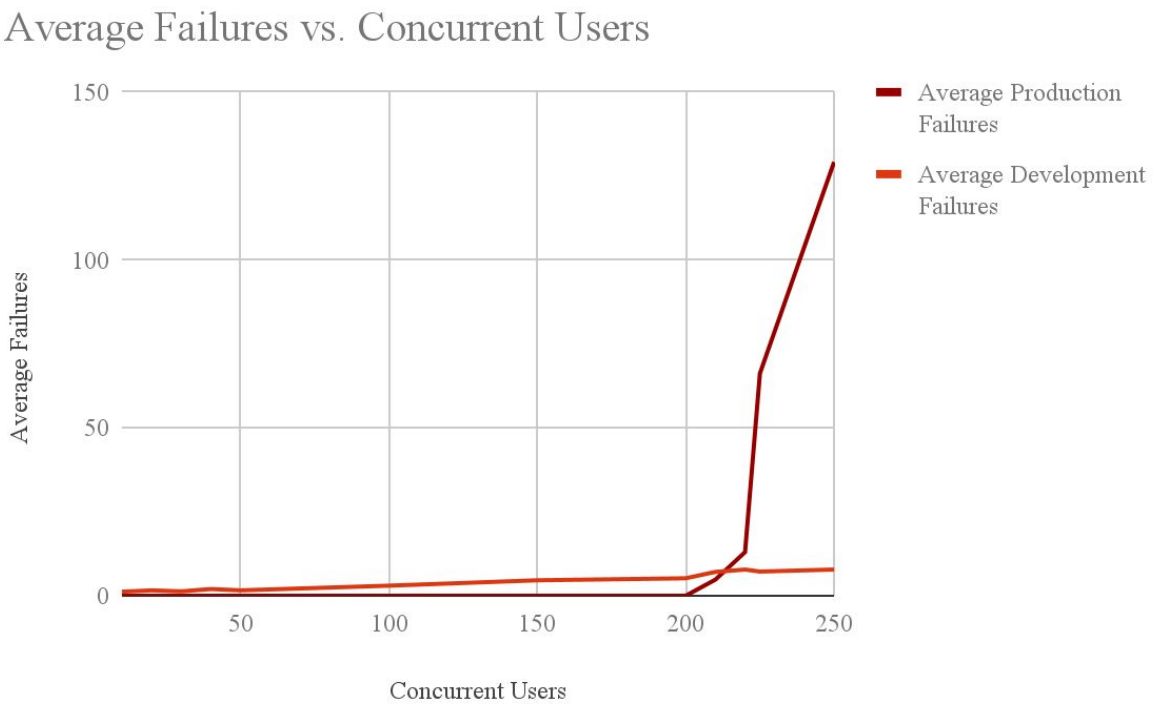


As is shown, 100% of requests were successful, and all were completed in 727 ms. There were no failures. Five subsequent tests were run with identical parameters, as was done with all future parameters, and all returned 100% successful. The project requirements have been successfully satisfied by this production server.

It is important, however, to test the server to its limits. Those results can help when deciding when to scale up on servers if the application attracts more users than anticipated. Tests were run up to 200 concurrent users, all of which returned 0 failed requests across five trials each. All signs indicate that the server can handle up to 200 concurrent users stably. After 200 concurrent users, however, that stability began to drop off. 210 users produced 3, 4, 0, 9, and 8 failures respectively. 220 concurrent users produced 25, 27, 1, 6, and 6 failures. 225 users produced 224, 32, 36, 21, and 18. This drop-off continued to its steepest failure at 250 users, which resulted in 249, 249, 12, 55, and 81 failures.

As a point of comparison, the development servers that we tested on consistently dropped between 1-5 requests when up to 200 concurrent users were uploading photos. After this, the average failure rate hovered around 7. So, while there is a case to be made for fewer drops in the 200-250 concurrent user zone, there is an element of unreliability in the 1-200 that the production server avoids with 0 failures. At this early stage, it would make more sense to trust

the production server to flawlessly carry the fledgeling user base we plan to have rather than imperfectly supporting users who may not be visiting this early.



With this data, the team would need to scale up after production if and when the application expects more than 200 concurrent users. After that, the production server becomes unstable. Either the hardware would need to be replaced with more powerful equipment, or subsequent servers would need to be spun up to share the load.

User Documentation

The majority of user documentation is spread out across code files and other parts of this document. Please see the sections below for references to where this documentation exists.

End User Help:

See section: User Interface Design under Application Design

Guide being displayed to user:

https://docs.google.com/document/d/1PCgq9_49LYgfW_xYmjJ_YyPvWE-r7yY_Mj9o5gx1Zrc/edit?usp=sharing

Back-End User Help:

See below section: Back-end admin tasks (only admins may access our back-end)

Administrator Users Guide:

Front-end admin tasks:

- See section: Administrative System Report Page under Application Design

Back-end admin tasks:

- See sections: Database Server
- API administrative routes may be called directly by an admin. Please see details in section: API Server, and in the adminRoutes.js file.

Code Maintenance Guide:

Read more on how everything fits together under section: Architecture

Read more on setup and run procedures under sections: Webserver, API Server, Stylizer and Style Server, and Database Server.

Code files are thoroughly documented for best maintainability. Each file has a header explaining what the file's purpose is, the majority of the code is self-documenting, and significant code lines are explicitly documented. Hopefully these are sufficient for future developers with a good CS background to work off of.

The project structure is the following:

- The api folder contains all of the Node.js Express middleware that serves the API.
- The web folder contains the Angular front-end project, which is the webpage that front-end users interact with. All website code is located in this directory.
- The stylizer folder contains the Express style server, which is what communicates with the central API.