PixColor: a web application to colorize pictures

Authors[i]: Toni Pohl and Cédric Bhihe
Affiliation at the time of writing: UPC – FIB / MIRI Program
Object: Cloud Computing (CLC) class project as part of the MIRI Master program requirements


[ii]

Foreword

The coder is Toni. I, Cedric, am in essence the scribe. That said we worked together and it was a pleasure doing so. Toni is a well of knowledge and I sometimes managed to articulate pertinent questions to draw water from that well. This report was born from that very lopsided collaboration.

i {toni.pohl,sedric.bhihe} @ est.fib.upc.edu
ii First page pictures were extracted from https://richzhang.github.io/colorization/

1. Introduction

1.1 – About this report

This project report takes the form of an hands-on tutorial, designed to guide you, as you build your first web application to colorize b&w snapshots. It relies on Free and Open-Source Software (FOSS), as well as on the freemium API offered by [Algorithmia](https://algorithmia.com) to colorize pictures.

Students of the subject may also use this report as a Beginner's Guide and a fast pace introduction to some of the technologies used in the project. The end result, the application we dubbed "***pixColor***", is the pretext that will allow you to become a little more familiar with modern web app programming concepts (e.g. REST[i], CGI, WSGI[ii]) and free resources in the form of APIs, repositories (GitHub), and other FOSS technologies readily available on the web. That includes Docker, Flask, Python and, to lesser extent because it is outside the scope of this report, AngularJS and Bootstrap CSS.

1.2 – About the web application

As you will see, simply colorizing a b&w picture using the appropriate Algorithmia API[iii] and any image previously uploaded to Algorithmia's web site represents a staggering 15 lines of Python code, including blank lines. The API gives access to a processor based on a deep learning semantics classification algorithmic model, developed by Zhang et al [1] to colorize 2D grayscale and legacy black&white objects. The resulting color images are also stored by default by Algorithmia. This requires the API user (i.e. Algorithmia's customer) to:
• create and log onto a private Algorithmia account,
• upload an image or an array of images in a dedicated Algorithmia repository,
• run a short Python program locally to invoke the API and trigger the remote image-processing routine. That requires proper authentication via a private key giving you exclusive access to your API-usage credit.
• visualize the colorized result, placed by the API in a dedicated storage on Algorithmia's site. This is done downloading the result file with your favorite image viewer.

Instead you may choose to set up a web interface on your local browser, where you may upload any b&w image, have it colorized and displayed at the click of a button. This however requires the implementation of:
• a web frontend GUI (html + JavaScript + CSS), to Base64 encode and decode the image
• a backend handling calls and I/O (Send/Retrieve) between the API and the frontend.
• the API provided by Algorithmia. In the remainder of this report, we treat the API as a grey box, with some documentation available on GitHub[iv].

1.3 – About Algorithmia (https://algorithmia.com/)

The driving force behind the business model of Algorithmia, Inc., based in Seattle, WA (USA), is that a lot of algorithmic intelligence is lost to the world as it is being developed. As developers, researchers and companies produce all kinds and types of functional algorithms, those mostly remain either unknown, under-utilized and without market value for lack of a proper marketplace to commercialize them. This is where Algorithmia, Inc. offers its services, Algorithmia is a marketplace for algorithms and provides a platform for their use, development, and commercialization as well as their deployment. We already knew Iaas, PaaS and SaaS. We now know ***AaaS***.

How it works:
Upon visiting the company's web site, open an account and secure a HASH, which will be your private API key. That key will grant you access to algorithms in Algorithmia's database. Should you choose to implement one, it will let you

i For more on RESTful web services, see https://en.wikipedia.org/wiki/Representational_state_transfer
ii For more on WSGI (Web Server Gateway Interface) specs, see https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
iii Algorithmia colorization API: deeplearning/ColorfulImageColorization/1.1.5
iv https://richzhang.github.io/colorization/

run that algorithm via its API on Algorithmia's infrastructure. The service cost is tiered; algorithms are proprietary and closed source.

Monetizing of the service by Algorithmia is based on a 3-tier model:
- "**Free-forever**": very small usage, typically by academics or lone developers – uniquely identified by their API key - are granted a free monthly credit of 5000 CPU cycles, plus 5000 CPU cycles on the first month, so they can experiment with the service.
- "**Pay-as-you-go**": medium usage with credit auto-reload (automatic purchase to replenish credit) and volume discount.
- "**Enterprise**": Premium service with personalized CPU&GPU architectures, dedicated algorithmic repo, tech support.

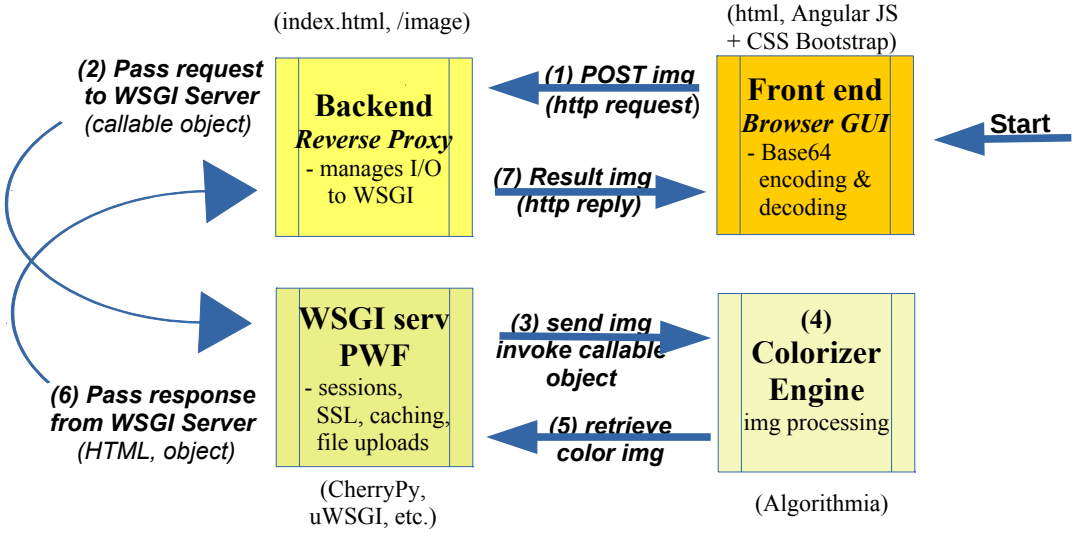
Credit consumption varies linearly with the amount of CPU time a client uses up- and down-loading images, as well as processing them. One CPU-second is worth one unit of credit. At the time of writing, one unit of credit costs USD 0.0002 and the smallest purchase possible is for USD 20 (i.e. 200,000 credits).

After a few minutes of hands-on practice, something interesting transpires from such treatment. Unsurprisingly up- and downloading large image-files costs more (in terms of CPU cycles) than doing so with smaller images. However customers have no *a priori* knowledge of how CPU-efficient their chosen algorithm is for a given data set to be processed. They can only interact via an API with an otherwise opaque algorithm, and algorithms are not ranked by Algorithmia from a CPU-efficiency perspective. For that reason customers will remain unaware that algorithm A may perform with more CPU-efficiency than algorithm B, written by someone else, lest they embark on a costly benchmarking effort. Still more tantalizing is the fact that any given algorithm may prove comparatively efficient for certain types of data and not for others depending on a number of proprietary technical implementation choices made by the authors of the algorithm.

Once their credit is exhausted, customers have no option other than purchasing new credit to keep using algorithmic resources marketed by Algorithmia. This, in a nutshell, is a replication of pre-paid telephone service model for AaaS.

1.4 – About the programming scheme

To develop a web app, one may adopt a standard server / web-app interface specification as a logical first step. Such is the role of the WSGI spec (PEP 0333 and PEP 3333), based on the Common Gateway Interface (CGI) of earlier WWW days; it presents a universal, server-agnostic interface to ensure maximum portability of web-apps developed in Python. In this sense it can be seen as a programmatic API, which sets rules for the development of Python web-apps. WSGI is a standard interface and Python Web Framework (PWF) that modules and containers can implement. It is now accepted and a major go-to approach for running Python web-apps. It relies on an underlying web server communication protocol layer, which ensures the correct handling of client requests as they are passed from the web server to the Python web-app. WSGI offers flexibility to developers and scalability of web traffic for dynamic content.



To ease development, we opted instead for the simpler scheme of a ***RESTful web service architecture*** [2]. REST (REpresentational State Transfer) imposes architectural constraints, among them that of presenting a uniform interface to access web resources through their textual representation (URI, URL) over the HTTP protocol [3]. Another constraint is that data communication between the web-server (Flask in our case) and an API-based set of tasks relies on the HTTP protocol, through the explicit use of a limited set of predefined HTTP methods, stateless operations also called *HTTP verbs*[i]. The best known among HTTP verbs are GET, POST, PUT, DELETE, OPTIONS. Quoting from an excellent IBM tutorial: " HTTP GET, for instance, is defined as a data-producing method that's intended to be used by a client application to retrieve a resource, to fetch data from a web-server, or to execute a query with the expectation that the web-server will look for and respond with a set of matching resources."

In the context of web-apps development, beyond a uniform interface, REST's principles include:

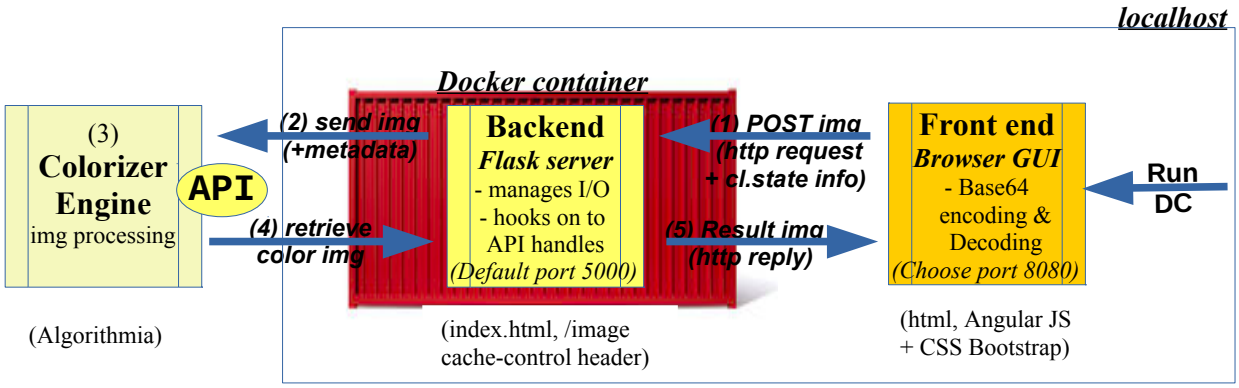
- a clear-cut client-server separation, where business logic + data storage (Backend) and user interface (Frontend UI) are effectively separated. This favors portability of the UI, simplicity and scalability, as well as the possibility for different Backend components to evolve independently from one another, a property of significance in distributed computing environments, or for instance in the context of SoA/microservices.
- the property of a layered architecture, i.e. of transparently accepting intermediaries (gateways, FW, proxies) unbeknownst to the requesting Clients.
- the absence of client state information being stored by the Backend

The Backend must rely exclusively on the state information provided punctually the clients at each one of their requests. An exception is when client state is transferred to another service and/or can be "persisted" by storing it temporarily in a dedicated database, e.g. for the purpose of authentication. In other words a client request being processed by the Backend doesn't require the Backend server to retrieve any kind of application context or state. All that is needed to process a request is already contained in that request's HTTP header and body.

We refer the reader to one of many online tutorials on RESTful implementation of web-apps, for more information on:

- how to address resources,understanding available MIME types for headers (XML, XHTML, JSON)
- formats of HTTP- based RESTful requests and responses

For ease of deployment we used the containerized scheme below:



i https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

On communication and port forwarding for dockerized single-page web-apps:

In the context of single-page web-apps Front- and Backend execute in the container environment on the same host, while the API RunTime is remote. The GUI (Frontend) is supported by the user's browser. While intercontainer communication is not an issue, communication which takes place between the outside world and the Backend (inside the container) meets with a container wall. Docker acts as a NAT for the Backend's exposed port(s). In our case flask's exposed is 5000 by default. For the container NAT to control packet forwarding and port translation, mapping of Backend port 5000 to GUI port 8080 (for example) must be specified by the user at the time the Docker container is created from its image.

Packet forwarding also occurs back and forth between the remote application engine (in Algorithmia) behind the API and the Backend. However in this case the API handles all necessary communication transparently and securely.

Meanwhile we need Docker NAT IPtables to allow all outgoing packets from Docker and any incoming packets to Docker, provided they are "answer-packets". Since at least Docker version 1.4.1 solves the packet-forwarding issue between API and Backend, upon container instantiation and for simple cases such as our single-page web-app. It does so by adding ad-hoc rules in the local host's IPtables. Later in this report, we will check the FORWARD chain of the IPTables to that effect.

2. Getting Started: install ressources

Follow instruction while on a computer, preferably running Linux. You should have `sudo` rights to be able to install packages on your local host. This tutorial does not provide instructions for proprietary platforms such as Microsoft Windows of Apple Mac OS. It is however possible to achieve identical result on those platforms.

We assume readers have prior knowledge of the Linux command line interface (cli) and bash and know how to interact with GitHub using `git`.

2.1 – Install Python 2.7.x

Make sure that Python 2.7.x is correctly installed on your system. In Linux terminal, type:

```
$ which python; python -V
```

If the response does not point toward Python v2.7.x, install either standalone Python 2.7.13 or Anaconda2, provided you have not arleady installed Anaconda3. Caution: installing Anaconda 2 on a system already equipped with Anaconda3 may result in unstable behavior. The code we present runs unmodified only on Python 2.7.

If you do not have Python 2.7.x installed on an older Debian based system and after determining that installing it will not cause problems, do:

```
$ version=2.7.13
$ mkdir -p ~/Downloads/; cd ~/Downloads
$ wget https://www.python.org/ftp/python/$version/Python-$version.tgz
$ tar -xvf Python-$version.tgz
$ cd Python-$version; ./configure; make; sudo checkinstall
```

The last command `` `sudo checkinstall` `` (instead of the more usual `` `sudo install` ``) will make it easier to uninstall when needed. Running earlier version of Python 2.7 may require some modification of the provided python script.

On more recent systems, just do:

```
$ alias sag='sudo apt-get'
$ sag update; sag upgrade -y; sag install python2.7
```

```
$ which python; python -V          # check on correct installation and version number
```

2.2 – Install and run Docker 17.03

Docker is a well documented FOSS project that automates the deployment of applications inside software containers. A Docker container (DC) is a set of restricted namespaces (a process namespace, a filesystem namespace, etc) where processes can run. Docker implements at its core a separation between application (and any dependencies needed to run it) from the operating system itself. To make this possible Docker uses *containers* as running instances of *images*. A Docker image is basically a stand-alone, executable template for a filesystem, including code, RTE, libraries, and config files. When you run a Docker image, an instance of this filesystem is made live and runs on your system inside a DC. By default this container cannot mess with the original image itself or the filesystem of the host where Docker is running. It is a self-contained environment.

In this section, you will learn how to:

- install Docker and to run it, pulling images,
- make your own Docker images and push them to DockerHub, the Docker registry or repository (repo).

After completing this section, you will:

- understand most of the terminology used in the Quick Start section of the Readme.md file in the GitHub repo https://github.com/Cbhihe/pixColor .

In a series of tutorials, Docker offers a more in-depth presentation for beginners to get started[i].

2.2.1 -- *Installation of Docker*

Installation on your platform is not mandatory for the application ***pixColor*** to run well. However it will simplify your configuration tasks and reduces the amount of scripting work, you will need to do to get everything rolling. Following this section will also equip you with useful information and hands-on experience about how to put Docker to use.

At the time of writing the latest Docker version was v17.03. Before installing the Docker Community Edition (*Docker CE*) , make sure you comply with prerequisites. When those prerequisites are satisfied, do from terminal:

```
$ alias sag='sudo apt-get'
```

In case older versions of Docker are present, remove them:

```
$ sag remove docker docker-engine
```

Update the list of available packages:

```
$ sag update; $ sag upgrade -y
```

Install extra packages to allow Docker to use the `'aufs'` storage drivers:

```
$ sag install linux-image-extra-$(uname -r) linux-image-extra-virtual
```

Set up Docker's repositories and install (here, the amd64 compatible version) from them, for ease of installation and future upgrades:

```
$ sag install apt-transport-https ca-certificates curl software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo apt-key fingerprint 0EBFCD88  # Verify the official GPG key fingerprint
  pub   4096R/0EBFCD88 2017-02-22
  Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
  uid                  Docker Release (CE deb) <docker@docker.com>
  sub   4096R/F273FCD8 2017-02-22
```

i https://docs.docker.com/get-started/

```
$ sudo add-apt-repository "deb [arch=amd64] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Update the list of available packages:

```
$ sag update; $ sag upgrade -y
```

Install the latest version of Docker:

```
$ sag install docker-ce
```

Or, install a specific version of Docker, if it suits your needs better:

```
$ sag install docker-ce=<VERSION>
```

To to avoid having to type `sudo` whenever you run the `docker` command, add your username to the docker group:

```
$ sudo usermod -aG docker $(whoami)
```

To add a user other than you to the docker group, declare that <username> explicitly:

```
$ sudo usermod -aG docker <username>
```

At this point, it is advisable to perform a full log out from session followed by a log in.

In the following sub-section, we assume that the user is part of the `docker` group. If not, prepend all commands with `sudo`.

Finally, to uninstall Docker:

```
$ sudo apt-get remove --purge docker-ce
$ sudo rm -rf /var/lib/docker
```

2.2.2 – ***Getting started with Docker***

Understanding the structure of the command (cmd) line in Docker is paramount.

First let us verify that Docker is (i) running) and (ii) installed correctly. First check that the daemon runs; second run the `hello-world` DC image.

```
$ /etc/init.d/docker status
  docker start/running, process 17930
$ docker run hello-world
  ...[output here]...
```

Using `docker` consists in passing options to it, then a command followed by arguments. The syntax takes this form:

```
$ docker [opts] [cmd] [args]
```

To view all available sub-cmds, type:

```
$ docker
```

To view the switches available to a specific sub-cmd, type:

```
$ docker <docker-sub-cmd> --help
```

For instance, to learn about how to use `build`, type:

```
$ docker build --help
  Usage:    docker build [OPTIONS] PATH | URL | -
  Build an image from a Dockerfile
  Options: ...
```

To view system-wide information about Docker, use:

```
$ docker info
```

2.2.3 – *Working with Docker* starts with understanding that Docker containers are run from Docker images. The rest of this sub-section is essentially a light rewrite of Digital Ocean's how-to[i] on the subject.

By default, Docker pulls images from its repo, DockerHub, a Docker registry managed by Docker. Anybody can build and host their Docker images on DockerHub. Most applications and Linux distros you'll need to run with Docker containers have images hosted on DockerHub.

You can search for images available on DockerHub by using the `docker` cmd with the `search` sub-cmd. For example, to search for the Ubuntu image, type:

```
$ docker search ubuntu
  ...[output here]...
```

The script will crawl DockerHub and return a listing of all images whose name match the search string `'ubuntu'`.

In the OFFICIAL column of the output, "OK" indicates an image built and supported by the company behind the project. Identify the image you need, and download it to your computer using the `pull` sub-cmd:

```
$ docker pull ubuntu
```

After downloafing an image of interest, run a container using that image using the `run` sub-cmd. If an image has not been downloaded when `docker` is executed with the `run` sub-cmd, the Docker client will first download the image, then run a container using it:

```
$ docker run ubuntu
```

To see the images that have been downloaded to your computer, type:

```
$ docker images
```

Images that you use to run DCs can be modified and used to generate new images, which may then be *pushed* to DockerHub or other Docker registries.

2.2.4 – *Running a Docker container (DC)* is as easy as what you already did at the beginning of subsection 1.2.2.

```
$ docker run hello-world
  ...[output here]...
```

In the above, a DC ran (very briefly) and exited immediately after emitting a message.

Containers can be run in interactive (attached) or non-interactive (detached) mode. To exemplify this, we now run an attached DC using an image of the latest Ubuntu operating system. The combination of the **-i** and **-t** switches gives you **interactive shell** access with a **virtual tty** and **capture stdin** into the container[ii]:

```
$ docker run -it ubuntu
  Unable to find image 'ubuntu:latest' locally
  latest: Pulling from library/ubuntu
  b6f892c0043b: Downloading  2.85 MB/46.89 MB ...
  55010f332b04: Download complete
  ...[more output here]...
root@d9b100f2f636:/# uname -a
  Linux 2a3747ea8be5 4.4.0-75-generic #96~14.04.1-Ubuntu SMP Thu Apr 20 11:06:30
  UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
root@d9b100f2f636:/#
```

The command prompt changed to reflect the fact that you're now working inside the container. Note that the **DC id** `d9b100f2f636` is in the new command prompt.

i https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04

ii https://docs.docker.com/articles/basics/#running-an-interactive-shell

You may now run any command inside the DC. For example, let's update the package database inside the container and install `nodeJS`. There is no need to prefix any command with `sudo`, because you are operating inside the container with root privileges:

```
root@d9b100f2f636:/# apt-get update
root@d9b100f2f636:/# apt-get install -y nodejs
```

To stop an active container from within its environment (i.e. from inside the DC), just exit it:

```
$ root@d9b100f2f636:/# exit        # to stop an active container in interactive mode
$
```

As already mentioned, one may also run container images in detached (non-interactive) mode, by including the **-d** switch. This allows USER to fall back to her underlying session while the container keeps running:

```
$ docker run -d -it ubuntu
  746fd4aceacffe3a60cc835ce95c97377884d2a96f23ad519c2e40de6fdc395f
$
```

To re-attach the DC, do:

```
$ docker ps [-a|l]          # to list containers: -a = all (active and inactive), -l = latest created
CONTAINER ID   IMAGE   COMMAND     CREATED     STATUS    PORTS   NAMES
746fd4aceacf   ubuntu  "/bin/bash" 19sec ago   Up 18sec          heuristic_archimedes

$ docker attach --sig-proxy=false 746fd4aceacf       # See Docker docs[i] for more info
root@746fd4aceacf:/#
```

To detach again the DC ***from inside it***, simply issue CTRL+p CTRL+q to recover your usual host shell-prompt neither killing the container nor any process running in its virtual environment.

To stop an active DC running non-interactively (detached mode) from the normal shell prompt, simply issue:

```
$ docker stop container-id          # container-id = 746fd4aceacf
```

or `$ docker stop container-name` # container-name = heuristic_archimedes

2.2.5 – *Creating a Docker image* consists in committing changes in a Docker container to a file. It is recommended, if you mean to preserve for future re-use any configuration work made in the DC's virtual environment.

After installing `nodeJS` inside the previous Ubuntu DC, you now have a running container, whose composition is different from that of the image you used to create it. To save the state of the container as a new image, first exit from it:

```
$ exit
```

Then commit the changes to a new Docker image instance using the following cmd:

```
$ docker commit -m "commit_message" -a "author" container-id repo/new_image_name
```

Specify: **-m** "commit message" that helps you and others know what changes you made,
-a "author".
container-id as noted earlier when starting the interactive docker session.

Unless you created additional repositories on Docker Hub, the repository is usually your DockerHub username:

```
$ docker commit -m "added node.js" -a "skb" d9b100f2f636 skb/ubuntu-nodejs
  sha256:a6f9c14d6ce51033b7798011b5e442848ed287b51d7e93332cb2716f7f58710d
```

When you *commit* an image, the new image is saved ***locally***. Later in this tutorial, you will push an image to a Docker registry like DockerHub, so it becomes publicly accessible.

i https://docs.docker.com/engine/reference/commandline/attach/#attach-to-and-detach-from-a-running-container

After the commit operation has completed, listing the Docker images now on your computer should show the new image, as well as the old one that it was derived from:

```
$ docker images
  REPOSITORY          TAG        IMAGE ID        CREATED          SIZE
  skb/ubuntu-nodejs   latest     a6f9c14d6ce5    2 minutes ago    206 MB
  ubuntu              latest     ebcd9d4fca80    24 hours ago     118 MB
  hello-world         latest     48b5124b2768    4 months ago     1.84 kB
```

In the above example, `ubuntu-nodejs` is a new local image, derived from the existing ubuntu image from DockerHub. The size difference reflects the change as `nodeJS` got installed. To run a DC using Ubuntu with `nodeJS` pre-installed, use the new image:

```
$ docker run -it --name foobar_dc skb/ubuntu-nodejs
root@7d6db03013ba:/#
```

Note that the DC id above is different from before, as you now run the container image under a new parent-PID. Note too that we added the new option **`'--name'`** which allows you to choose a name for each DC instance you may want to create (here "foobar_dc"). That name may be used at your convenience to detach and re-attach DCs, commit changes, stop a DC, as you would for these and more operations, when using the DC id.

To rename a DC, issue:

```
$ docker rename old_name new_name
```

2.2.6 – *Building images from Dockerfile* is another convenient way to automate the build of an image and to include a context in it.

• Docker reads instructions from the `Dockerfile`, a text document which contains the commands a user could call on cli to assemble an image. Users can read command-line instructions from a Dockerfile located anywhere on the host, with: `$ docker build -f /path/to/Dockerfile`

• The context is the set of files at the build location, typically specified by `.` to signify "present directory and recursively parsed sub-tree". It can however be any directory and sub-tree specified by its path or a URL.

When no fully qualified path for `Dockerfile` is provided, `docker` will revert to the default Dockerfile location, which is the root of the context.

The content of the Dockerfile is parsed and run by the Docker daemon, not by the host's shell. It conforms to a set of specific syntactic rules and uses case-insensitive reserved words. The complete documentation on this subject is available at https://docs.docker.com. A Dockerfile is similar in concept to the recipes and manifests found in infrastructure automation (IA) tools like Chef or Puppet.

2.2.7 -- *Pushing DC images to a Docker repo* is the next logical step, after creating a new container image.

To push an image to DockerHub or any other Docker image repo, you must have an account on that repo.

To learn how to create your own private Docker repo[i], see How to set up a private docker registry on Ubuntu. In the remainder of this guide, we will only concern ourselves with pushing Docker images to DockerHub.

First create an account on DockerHub by registering on the Docker's website. In order to push a container image, you must have logged into DockerHub from your local terminal session. You'll be prompted to authenticate:

```
$ docker login -u docker-registry-username
  Enter password:_
```

i https://www.digitalocean.com/community/tutorials/how-to-set-up-a-private-docker-registry-on-ubuntu-14-04

Then you may push your own image using:

```
$ docker  push docker-registry-username/docker-image-name
  The push refers to a repository [docker.io/skb/ubuntu-nodejs]
  ...[output]...
```

After pushing an image to your GitHub repo, it should be listed on your account's dashboard.

2.2.8 – ***Miscellaneous***

For a refresher on how to use Docker, take a look at the excellent Docker Cheat Sheet on GitHub[ii].
For security minded users, references on how to get started in that direction can be found here, in relation with *Docker Compose,* an orchestration tool for Dcs.

2.3 – Install Flask

Flask, our light-weight Backend server, is installed as a module from the python wrapper.

Install a Docker container from a Dockerfile that already tends to the loading of the proper module for `flask` and `algorithmia`.

3. Dockerizing a Python Flask Application

3.1 – Clone the GitHub repo

On your host, go to the git directory where you wish to clone the GitHub repo ***pixColor***.

```
$ git clone https://github.com/Cbhihe/pixColor.git
```

Check out the master branch in your local repo `pixColor` and verify its content:

```
$ cd pixColor
$ git checkout master; \ls -Afl
  total 24
  -rw-rw---- 1 ckb developer  144 May 20 19:10 Dockerfile
  -rw-rw---- 1 ckb developer   10 May 20 19:10 .dockerignore
  drwxrws--- 8 ckb developer 4096 May 20 19:10 .git/
  -rw-rw---- 1 ckb developer 2088 May 20 19:10 main.py
  -rw-rw---- 1 ckb developer 1657 May 20 19:10 Readme.md
  drwxrws--- 2 ckb developer 4096 May 20 19:10 static/
```

Visualize the content of `Dockerfile` and of `main.py`:

```
$ vi Dockerfile
  FROM python:2
  RUN mkdir -p /opt/app
  COPY . /opt/app
  RUN pip install algorithmia flask
  WORKDIR /opt/app
  # Specifies which port should be made available for mapping by Dockerfile
  # Default Flask communication port 5000
```

ii https://github.com/wsargent/docker-cheat-sheet

```
EXPOSE 5000
ENTRYPOINT python main.py
```

Listings of `main.py` and `index.html` (inside the `static` directory) are available in Appendices A and B. We think it is important to understand the programming scope of `main.py`. Comments are provided throughout the code, so the beginning student can better understand the structure of the program.

Inversely `index.html` written in html with Javascript and Bootstrap CSS falls outside the scope of this report. Because Bootstrap CSS introduces remote dependencies, our web-app needs a few seconds to load them, before the GUI (index.html) can be fully interpreted and rendered in your browser pane.

3.2 – Build an Docker container image

Build a Docker image from the Dockerfile:

```
$ docker build -t CONTNR_NAME:CONTNR_TAG .
```

The "-t" flag above adds a tag to the container image so that it gets a nice repository name and tag. Note the final ".", which tells Docker to use the Dockerfile in the current directory. The build may take a while installing necessary resources per your Dockerfile. After build completion, to see the effect each dockerfile action had on the overall size of the resulting image, run:

```
$ docker history CONTNR_NAME:CONTNR_TAG
```

Check final image size with:

```
$ docker images
  REPOSITORY      TAG           IMAGE ID            CREATED            SIZE
  colorize        dockerfile    e41f8a38893c        1 minute ago       688 MB
```

where in this example CONTNR_NAME is “colorize” and CONTNR_TAG is “dockerfile”.

3.3 – Run a container image and launch the GUI on your browser

Finally create an container and run it with the proper port mapping for GUI and backend to be able to communicate:

```
$ docker run -d -p 8080:5000 -e API_KEY=YOUR_API CONTNR_NAME:CONTNR_TAG.
```

Check that the NAT IPtables *`FORWARD`* chain is correctly configured by Docker:

```
$ sudo iptables -vnL FORWARD
  Chain FORWARD (policy DROP 0 packets, 0 bytes)
pkts bytes target  prot opt in      out      source        destination
0     0    ACCEPT  all  --  *       docker0  0.0.0.0/0     0.0.0.0/0  ctstate,RELATED,
ESTABLISHED
0     0    ACCEPT  all  --  docker0 !docker0 0.0.0.0/0     0.0.0.0/0
0     0    ACCEPT  all  --  docker0 docker0  0.0.0.0/0     0.0.0.0/0
```

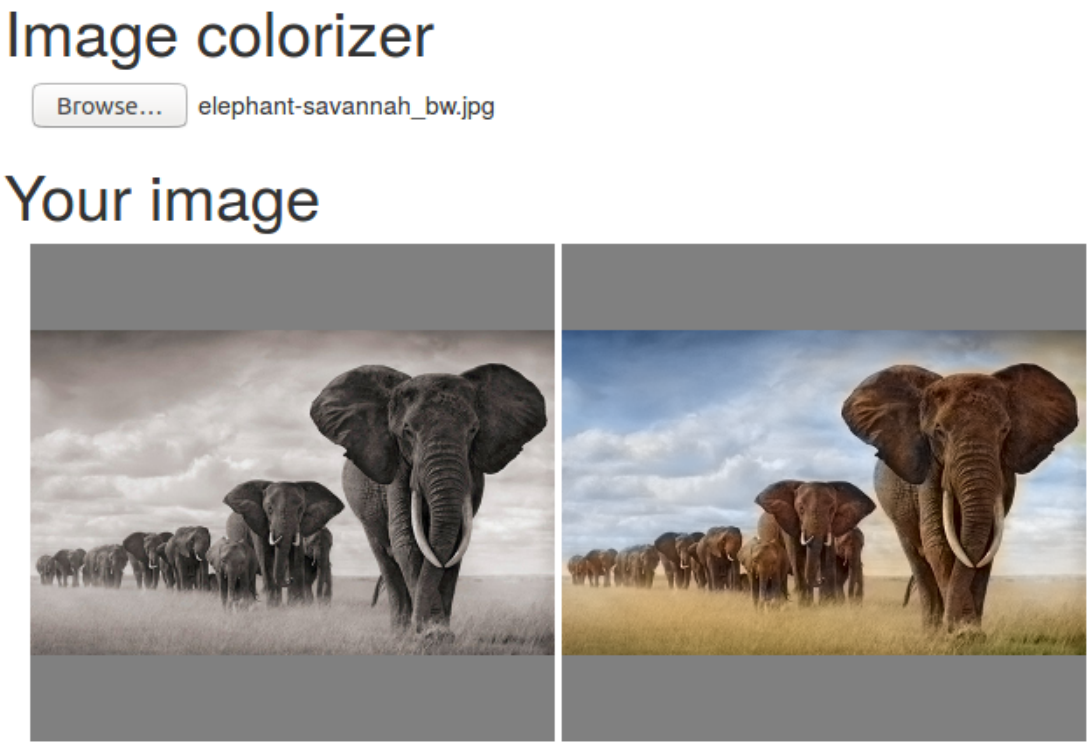
The output should contain the above 3 lines in the same order they are listed. Starting from the last :

- The third rule is docker→ docker intra-container communication, which is accepted without restrictions.
- The second rule is docker→ anywhere(but not docker), which is also accepted without restrictions
- The first rule is anywhere→ docker, only "answer" packets get accepted.

Unless outgoing filters are needed, the above is satisfactory as a minimum requirement for a one container web-app. Note tat those rules circumvent Your Ubuntu host’s `ufw` rules if there is a container listening. `ufw` does nothing in the NAT IPtables as Docker sets its rules higher up in the FORWARD chain. For that reason it is not possible to block an IP address or do rate limiting of any kind. To satisfy the latter requirement a few possibilities exist. A basic solution would entail running Docker with the switch `--iptables=false`. The upshot in that case is that all IPTables must be configured manually.

3.4 – Result

In a new tab in your browser, type in `0.0.0.0:8080` or `localhost:8080`. Upload a b&w image on disk using the browse button in the GUI and wait for the colorized result. As shown below the colorized image is automatically displayed in your browser.



APPENDIX A (`main.py`)

```
""" main.py """

import os
import base64
import Algorithmia

from flask import Flask, request

APP = Flask(__name__, static_url_path='')

API_KEY = os.environ.get('API_KEY')
# API_KEY is provided by Algorithmia on account registration
# API_KEY must be set as an environment variable in the app's RTE
if API_KEY is None:
    raise Exception("Environment variable API_KEY is not set.")

# Object construction
# instantiate Algorithmia module class for wanted algorithm
CLIENT = Algorithmia.client(API_KEY)
ALGO = CLIENT.algo('deeplearning/ColorfulImageColorization/1.1.5')

def process_image(image_base64):
    """ API call """
    req = {
        "image": image_base64
    }
    # returns result of HTTP request passed on to flask
    return ALGO.pipe(req).result

def get_image_base64(image_file):
    """ image encode base64 """
    return "" + base64.b64encode(image_file.read())

# Only accessed upon loading the flask site frontend
# GET received by backend when frontend is loaded
# Backend returns index.html to frontend (GUI)
@APP.route('/', methods=['GET'])
def root():
    """ ___  """
    return APP.send_static_file('index.html')

# Invoked when input file (b&w) is passed to backend ...
@APP.route('/image', methods=['POST'])
def image():
    """ ___ """
    # flask magic here !
    result = process_image(request.data)
    # "data" is an attribute of the "resquest" object in flask
    # "request.data" contains input image as large base64 encoded string
    # passed to main.py by frontend.
    return get_image_base64(CLIENT.file(result['output']).getFile())

# Algorithmia gets processed file from loc "output" (has URL form)
# where "CLIENT." specifies API_KEY and "getFile()" downloads binary file
# to be base64 encoded by frontend.

if __name__ == '__main__':
    APP.run(host='0.0.0.0')
    # not just processes in container but any host incl localhost (127.0.0.1)
        # can access the website served by flask.
```

APPENDIX B (`index.html`)

```
<!--
The following code provides a frontend to an Algorithmia's API based application, designed to
colorize black and white or grayscale pictures. It is part of a CS project, executed to meet
requirements of the MIRI Master Program at UPC in Barcelona, Spain.
The frontend of the project was built with AngularJS and Bootstrap.
Toni Pohl wrote both the backend and frontend code in a couple of hours, in April 2017, at the time
he was a MIRI visiting student.

Permission to fork or copy the code is hereby granted, provided that:
    you always mention that it is a fork from repo https://github.com/k4l4m/CLC-Project-Colorize.git
    you cite Toni Pohl as the original author of the forked code.
-->

<!doctype html>
<html ng-app="imageColorizer">
<head>
   <title>Image colorizer</title>
   <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">
</head>
<body>
   <div class="container-fluid">
      <div class="row" ng-controller="UploadController">
         <div class="col-md-6 col-md-offset-3">
            <h1>Image colorizer</h1>
            <div class="col-md-12">
               <input type="file" id="filechooser" ng-model="imageFile"
onchange="angular.element(this).scope().fileNameChanged(this)">
            </div>
         </div>
      </div>

      <div class="row" ng-controller="ImageController">
         <div class="col-md-6 col-md-offset-3">
            <h1>Your image</h1>
            <div class="col-md-12">
               <canvas image-canvas base64="original" style="background-color: grey"></canvas>
               <canvas image-canvas base64="processed" style="background-color: grey"></canvas>
            </div>
         </div>
      </div>
   </div>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.6.1/angular.min.js"></script>
   <script>
      var app = angular.module('imageColorizer', []);

      // Image factory for unprocessed image
      app.factory("SelectedImage", function($q) {
         var imageDefer = $q.defer();

         return {
            set: function(imageBase64) {
               imageDefer.notify(imageBase64);
            },
            get: function() {
               return {
                  then: function(cb) {
                     imageDefer.promise.then(null, null, cb);
                  }
               };
            }
         }
      });

      // Image factory for processed image
      app.factory("ProcessedImage", function($q) {
         var imageDefer = $q.defer();

         return {
            set: function(imageBase64) {
               imageDefer.notify(imageBase64);
```

```
            },
            get: function() {
               return {
                  then: function(cb) {
                     imageDefer.promise.then(null, null, cb);
                  }
               }
            }
         }
      });

      // Controller for uploading image
      app.controller("UploadController", function($scope, $http, SelectedImage, ProcessedImage) {
         var fileReader = new FileReader();

         fileReader.onload = function() {
            $scope.$apply();
            SelectedImage.set(fileReader.result);

            $http.post('/image', fileReader.result).then(function(res) {
               ProcessedImage.set(res.data);
            }, function(err) {
               console.log(err);
            });
         }

         $scope.fileNameChanged = function(elem) {
            var imageFile = elem.files[0];
            fileReader.readAsDataURL(imageFile);
         }
      });

      // Controller for image
      app.controller("ImageController", function($scope, SelectedImage, ProcessedImage) {
         SelectedImage.get().then(function(imageBase64) {
            $scope.original = imageBase64;
         });

         ProcessedImage.get().then(function(imageBase64) {
            $scope.processed = imageBase64;
         });
      });

      // Directive for image canvas
      app.directive("imageCanvas", function() {
         return {
            retrict: 'A',
            scope: {
               base64: '='
            },
            template: '<canvas class="image-canvas"></canvas>',
            link: function(scope, elem, attrs) {
               var canvas = elem[0];
               var ctx = canvas.getContext("2d");

               canvas.width = 300;
               canvas.height = 300;

               scope.$watch("base64", function(newValue) {
                  if (newValue === undefined) return;

                  // Clear the canvas
                  ctx.clearRect(0, 0, canvas.width, canvas.height);

                  // Draw image scaled down to 300x300 pixel canvas
                  var img = new Image();
                  img.onload = function() {
                     var imgWidth = img.naturalWidth;
                     var imgHeight = img.naturalHeight;

                     var widthScale = imgWidth > imgHeight ? 1 : imgWidth / imgHeight;
                     var heightScale = imgWidth < imgHeight ? 1 : imgHeight / imgWidth;

                     // Draw actual image
                     ctx.drawImage(img,
```

```
                        canvas.width / 2 - 150 * widthScale,
                        canvas.height / 2 - 150 * heightScale,
                        300 * widthScale, 300 * heightScale);
                };

                img.src = newValue;
            });
        }
    };
  })
</script>
</body>
</html>
```