Cloud-based RAW image editing

Student Name: Ryan Collins (gcdk35)

Supervisor Name: Dr Tom Friedetzky

Submitted as part of the degree of M.Eng Computer Science to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

January 24, 2018

***Abstract* —**

Context/Background RAW images are used by photographers to improve the quality of their images, specifically when editing. With JPEG files, the loss of data from compression leads to poorer quality edits, while using RAW results in a higher dynamic range. The editing of these files is traditionally done on a high-spec machine, capable of processing images with large resolutions. In the field, this can be difficult. When used for websites, the standard process involves editing a RAW file locally and exporting it, uploading the exported image to a Content Management System (CMS). Here, the CMS will then edit the file again, resizing the image to ensure it displays properly on the browser.

Aims The main aim of this project is to test the feasibility of a Cloud-based RAW image editor, allowing image adjustments such as exposure adjustment, colour adjustment, and overall improvements to the quality of the image, all within a cloud application rather than on the desktop.

Method A render server back-end will first be implemented as an API, taking in an input as a JSON object, and then processing the image, and then a JavaScript client shall be created to interface with this API.

Proposed Solution A web application, using a Java server with dcraw and custom image processing code to process RAW images, communicating over a socket.io interface, and a web front-end that allows users to interface with the render server.

***Keywords* —** RAW image editing, dcraw, cloud image editing

I INTRODUCTION

Many photographers use a file format (or rather, a family of very similar file formats) called RAW, which rather than compressing the image and conducting some image manipulation on the camera, store the RAW camera sensor data outputted by the camera sensor, for later processing and editing on a computer. These files can be much larger than the compressed image, but provide a far greater degree of control over the captured image, when compared with a compressed JPEG, along with an increase in quality. A RAW file essentially acts as a digital negative, as the image can be edited constantly without losing any quality between edits. (Verhoeven 2010)

A Project Aim

The aim of this project is to produce a Cloud-based RAW image photo editor, both as an API (to allow interfacing with any web system we can choose), along with a full RAW image editor interface via a web browser. The image photo editor should be able to read at least DNG and NEF (Nikon's format) files, and allow contrast, colour and exposure adjustment, along with noise reduction, haze removal, and auto-correction options. Furthermore, this system should be multi-user, allowing more than one user to edit RAW photos at a time through the web interface, each with their own individual collection of images.

B Deliverables

The project shall have the following deliverables:

Minimum Objectives

- Exposure adjustment
- Noise reduction methods (Gaussian, mean, median)
- Web Interface interacting with an image processing server
- Non-destructive image adjustment (i.e. no reduction in quality over time)

Intermediate Objectives

- Load DNG RAW files by upload
- White Balance Adjustment
- Gamma Correction
- Modern, user friendly User Experience
- Cropping, Rotating and Exporting to other formats

Advanced Objectives

- Addressing potential scalability issues
- Haze removal

II DESIGN

This section outlines the design of the system, starting off with the specification of what such a system needs to have, followed by further research, options for designing different components of the system, along with some information on implementation detail. Furthermore, details on architecture are outlined here.

Number	Requirement	Priority
FR-01	Allow the user to edit the exposure of a RAW image	High
FR-02	Allow the user to adjust the colour saturation and hue	Medium
FR-03	Allow the user to export their edits to several popular file formats: JPEG, PNG, and TIFF	Medium
FR-04	Store the render settings of a given RAW file (as specified by the user), to allow for the image to be re-rendered and re-edited without losing quality	High
FR-05	Have the ability to edit user specified files (i.e. users can specify which file to edit, rather than using a hard-coded file within the system)	High
FR-06	The client interface should have the ability to zoom in and out of the preview image, to help aid editing	Low
FR-07	Allow the user to apply automated image enhancement algorithms to the image to find the best parameters for each image	Low
FR-08	Allow the user to set their own white balance	Medium
FR-09	Allow the user to apply gamma correction to their RAW image	Medium
FR-10	Allow the user to control the highlights of the image	Low
NFR-01	The system should be able to cope with at least 2 users simultaneously using the system	High
NFR-02	The system should allow the editor controls to be hidden, to show the preview image on its own	Medium
NFR-03	The render server should provide a preview within 15 seconds of changing a parameter	Low
NFR-04	The user interface must be accessible through a web browser	High

Table 1: Functional and Non-Functional Requirements

A Requirements

Table 1 shows the functional requirements of the project, which define the functional elements of the system being produced with the prefix FR. The non-functional requirements, which don't directly relate to the functionality of the system, but are performance-based attributes that ensure the system will be more likely to succeed at the aims, are denoted with the NFR prefix.

B Proposed Extensions

Despite the main functionality shown above, there is still some room for improvement. Here are the potential changes/additions that can still be made to our system to improve the overall effectiveness. While we likely won't get time to implement all (or perhaps any) of the following, these are improvements that can be made.

B.1 Quality of Service Improvements

Client Side Preview Rendering The system itself renders the RAW images entirely on the client side, directly from the RAW file. However, the time for a render request to be sent, and the preview to be returned from the server can be quite large. Currently, the user will just be required to wait, but an improvement can be made where simple operations can be carried out on the client side, using the

previous preview image as the starting image, and therefore there will be a gradual speedup due to the lack of network traffic or having to run computationally extensive tasks on the entire RAW file (simply a compressed version of the preview).

Message Queue Implementation Currently, jobs are served when they first appear, but this can be extended to store each job in a queue and service each of these using a preset number of render servers. This will better scale for a larger number of users, and would allow for better scalability of the system (just by adding more render servers) without much modification to the system being necessary.

B.2 Additional Features

Image Haze Removal In images, it's possible that haze can creep into the image from background sources. In order to combat this, single image haze removal can be implemented to remove haze using an automatic algorithm. Several different algorithms exist, some using colour attenuation, others using dark channel prior.

By using colour attenuation prior (using the colour attenuation information already known about the image), we can use the difference between the brightness of the image and the saturation, in order to estimate the amount of haze present at a particular point. This can then be used to produce a map of the depth of an image, from only one image, and then by applying the atmospheric scattering model in reverse, the haze-reduced image can be yielded. This also has much better running time compared with other similar algorithms, which is more suited to our system (as we need to reduce the time taken to render images as much as possible). (Zhu et al. 2015)

C Architecture

As the system is fairly large, it's important to break it down into individual pieces. These are:

C.1 Client Interface to Render Server Communication

There are several options for communicating between the user interface and the render server.

Representational State Transfer (REST) A Representational State Transfer system would work by sending a request to the server, requesting an image be rendered. This initial request will be replied to with a job number, which will be quoted in further communications. From here, future requests will be made over a regular interval, requesting the information for the specified job. If this job is finished, the result will be returned, but otherwise further requests will need to be made. This is the process of polling. The XMLHttpRequest object in JavaScript, which is used to make AJAX requests, was designed by Microsoft in 1999, and later adopted in the 2000s. This method is definitely the most compatible with browsers, being compatible with Edge, Chrome, Firefox, Internet Explorer 7+, Opera, and Safari. (Mozilla 2017)

However, this method does require making many requests and connections to the server, coupled with code to regularly poll at an interval until done. While this will work, it's not the most optimal solution, and the code produced from this would be more complex (code could be needed to issue jobs, recall jobs, and to ensure that the person who requests the result of a job is actually allowed to see the result of the job).

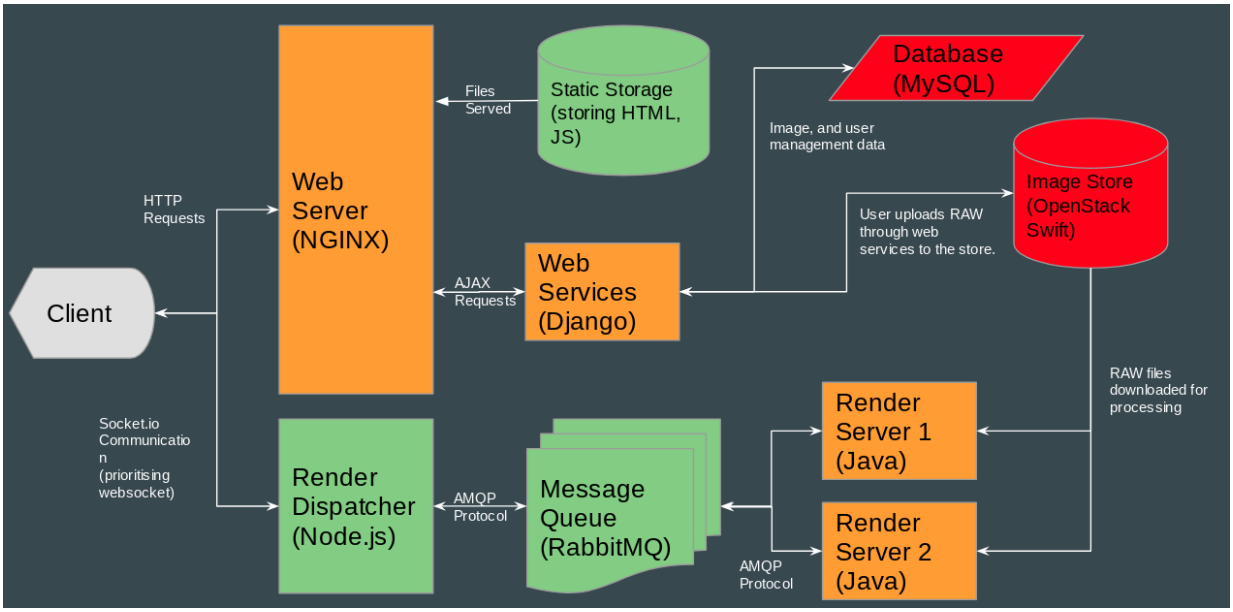

Figure 1: The architecture of the system. The user makes a request to the web server (reverse proxy). They fetch the static web application, and this sends requests (via AJAX) to the Web Services server behind the proxy, dealing with user login, file upload, and image management. When the image editor is loaded, the image information is fetched, and loaded in. Regularly, requests are made via Socket.io to the render dispatcher to render the image, and this is passed to the message queue for later computation (when the render servers are available). One of many render servers from the pool fetch the JSON string from the message queue, process the image using the associated settings, and return a Base64 encoded preview, along with other information. This is then passed back to the queue, to the render dispatcher, and finally via Socket.io to the user who requested the preview.

Web Socket Web Socket allows for two way communication between a browser and a server. It acts in a similar way to traditional TCP socket communication, only it incorporates the origin-based security model used within web browsers. By using web socket over REST, opening multiple HTTP connections is not needed, as a single connection is maintained at all times. (Melnikov & Fette 2011)

Socket.io Socket.io is an implementation that relies on both REST and Web Socket. If the browser supports web socket, then it is utilised, otherwise, it defaults to using REST and polls regularly for information. This way, we get the best of both options shown above, in such a way that the code itself remains fairly tidy (as the polling nature of REST is abstracted away from our system).

C.2 Render Server

The rendering server takes instructions given to it (in accordance with our API) and generates the output image based on the RAW image supplied, and the appropriate settings.

One of the first considerations is how to parse RAW files.

dcraw Dcraw is an executable that allows the processing of RAW image files.

Dcraw itself can then be used to convert to other formats, one being the uncompressed TIFF format,

giving us a very high-quality image that can then be adjusted. Our system isn't merely a wrapper for dcraw in this instance but extends the features supplied by dcraw.

Despite being an executable, it's written using only standard C libraries, and therefore it's fairly portable, not requiring any dependencies (aside from compiling with a C compiler, if no binaries are downloaded). (Coffin 2017)

According to the man-page, the executable contains commands to set RAW exposure, export as TIFF, set saturation level, set white balance, set colourspace, set gamma curve and flip the image. (Coffin 2015)

libraw LibRaw is a C++ library based on dcraw, that is designed as a library rather than just an executable.

LibRaw would be used when loading RAW images, processing them, and then from this, the image can be processed using custom routines (i.e. converting the libraw format to a matrix representation, and then using that matrix representation to carry out some manipulation).

While LibRaw has many useful features, the documentation is somewhat limited.

ImageMagick ImageMagick is a library that can be used to process any images, not just RAW. While it can read RAW image files, it also has many more features, including many features that we don't need/want to implement ourselves for the purpose of this project. As such, I believe while ImageMagick is a good option, it's a bit too heavy for our use.

C.3 Client-side Interface

The page design of the interface shall follow the design in D. The goal of the client side interface is to allow adjustment of the image parameters, and show a preview whenever a parameter is changed.

To display the image, an HTML5 Canvas is an ideal display vector for the previewed image, allowing us to implement features such as zooming, and drawing. This can't be achieved easily using a standard HTML image.

D User Interface

A sidebar should be used as the main interface for adjusting image parameters. This sidebar should be able to be hidden, showing the image fully underneath. When the sidebar is in the expanded state, the preview image should be displayed fully on screen.

The user interface design is shown in Figure **??**

Within the sidebar, clicking on a navigation item will display a new submenu. If the item is a parameter adjustment, updating the value in the menu will also update the value that is sent to the render server to generate a preview.

E Implementation Information

Each individual module of the system requires different technologies in order to produce an overall system.

Client-side Interface For this, HTML5, CSS3 and JavaScript (with jQuery to provide extra functionality) are ideal, as HTML5 can be used to create the user interface, with CSS3 providing styling and some basic animation to improve the UX. Using JavaScript with jQuery, it allows us to keep track of

the state of the editor, and transmit and receive information between the client and render dispatcher with help of a library. jQuery is an ideal library, providing many functions such as dynamic HTML loading an AJAX support, in a much more concise manner to using standard JavaScript.

As we are using mostly static content, only a few web services are needed for image selection and maintaining user uploaded RAW files, A web server like Apache/NGINX can be used to serve these static files, using server-side scripting only to determine whether a user is authenticated, and if so forwarding headers can be used to serve the static file without needing to serve static files through the scripting language itself, which creates an unnecessary workload and increases load times.

NGINX is what I'd recommend in this situation because unlike Apache, NGINX isn't configured out of the box for dynamic content, but Apache is configured to use PHP out of the box.

User account based RAW file management In order to select and upload images, a method of specifying users, and their uploaded images needs to be created. This will require a database, to store the user information (username, password), pointers to the images (image URL, and user associated with each image), along with some server side scripts to manage authentication, dealing with file uploads, and maintaining collections of images (listing all images associated with a user).

While any web framework would work with this, I've chosen to use the Python programming language with the Django web framework, as database queries can be made using the built-in Object Relational Mapper (ORM), that automatically writes SQL queries for the specified database back-end (e.g. MySQL, PostgreSQL), based on defining models as Python classes (inheriting the django models.Model class).

Furthermore, Django contains built-in authentication, and built in methods to create users, login, and managing sessions. These two features simplify the creation of the server side scripts for user account based RAW file management.

For the interface, rather than sticking to server side generated pages, serving a JavaScript based interface for this means the files can all be stored on the static file server used for the editor, and therefore AJAX requests can be made to the server to get the information needed, and load them onto the page. This avoids having to generate an entire page just for a few pieces of information. This JavaScript code shall make a request to get a list of the images (encoded as JSON, along with their associated urls). When a user clicks on the URL, the image picker loads up the editor, passing the URL to the image to edit (through GET variables). This way, there is less reliance on the entire server side framework.

In order to ensure all of this works, a database is needed. Django officially supports three databases: PostgreSQL, MySQL, Oracle and SQLite. SQLite is file based, and while it's OK for single user apps, for multiple users it won't scale well.

MySQL is far better for applications, implementing most (but not all) of the SQL functionality. However, it doesn't perform as well with concurrent users on write operations (e.g. writing new images into the database), or writing the associated image settings to the database. Furthermore, features are missing like full text search.

PostgreSQL is the most advanced, being fully standards compliant, and it has concurrency features built in to avoid using read locks. It's extensible, and allows for features like full text search. However, it can potentially be slower for primarily read based operations compared to MySQL, and as setup complexity for Postgres is far higher, it might not be the best tool for our project. (Tezer 2014)

Therefore, for this part of the system, Django shall be used alongside MySQL.

Render Server While lower level programming languages might be useful for image processing, the requirement of linking this with web technologies means that a language such as C or C++ is not as ideal.

Java is a better choice, as there are libraries available to provide web services (REST), along with Web Socket implementations, along with Socket IO. Furthermore, Java contains some libraries within the language that allow for image manipulation, most notably BufferedImage related features such as ConvolveOp, image resizing, and various other standard features build in. This is useful, as rather than starting completely from scratch, the custom image processing routines can build on the built in Java implementations, and create more advanced algorithms using them.

While Java doesn't support loading TIFF into BufferedReader using ImageIO directly, an extension exists online, in the form of TwelveMonkeys. No RAW image loading library exists within Java, but an executable such as dcraw or rawspeed can be used to generate a file that can be read within Java, while retaining the quality (e.g. uncompressed TIFF). This can be done using ProcessBuilder, and ideally writing a library to do this.

As this system will likely use a large amount of external dependencies, a dependency management tool like Apache Maven should be used, to both manage dependencies from an external source (e.g. GitHub, Maven Repositories), and also to build the system.

Account Management Server As this system needs to be used by multiple people, we need to have some record of users, and their associated images.

Rather than rendering each page dynamically, we can instead use JavaScript to make AJAX requests, fetching the data needed, and this way, we don't need to render whole HTML web pages. In terms of back-end technology, it doesn't matter too much what is used, as it won't be used too much, but in my case I'll be using Node.JS with an Express server, simply because it's easy to deploy and install dependencies ("npm install" can be run to install dependencies in one command), and it can also connect to a database to provide services such as login, and records of individual files.

File Storage Server As the user is required to upload files, and the files that are exported need to be stored somewhere (not on the render server local disk) to be accessed, it's important we have a centralised store.

Ideally, this file storage server should be interchangeable, allowing the user to change where the files are stored (only requiring it to be accessible by using a URL).

There are a few options on the public cloud for use with this system, namely Amazon's S3, and other storage services. However, as this prototype will initially be employed on a private cloud, I shall use a service called OpenStack Swift, which allows the storage of files. Furthermore, as this links in as a Django storage backend, and APIs already exist to tie Django in with Swift, this makes the implementation easier. As this will be separate from the web services server, and also separate from the render server, any loss of either of these will cause the files to remain online, and also allows for more user account management servers and more render servers to be placed online, fetching and writing files to this file storage server.

Distributed System Management As this system consists of several smaller systems, it's important that these are managed. For each smaller component, a Docker container can be created, containing the configuration between each one. From here, Docker Compose can then be used to manage the entire system, bringing everything online, opening ports, and building everything automatically.

This requires us to create a Dockerfile per system (one for the render server, one for the client front-end). This Dockerfile specifies the image to build from, along with instructions that need to be

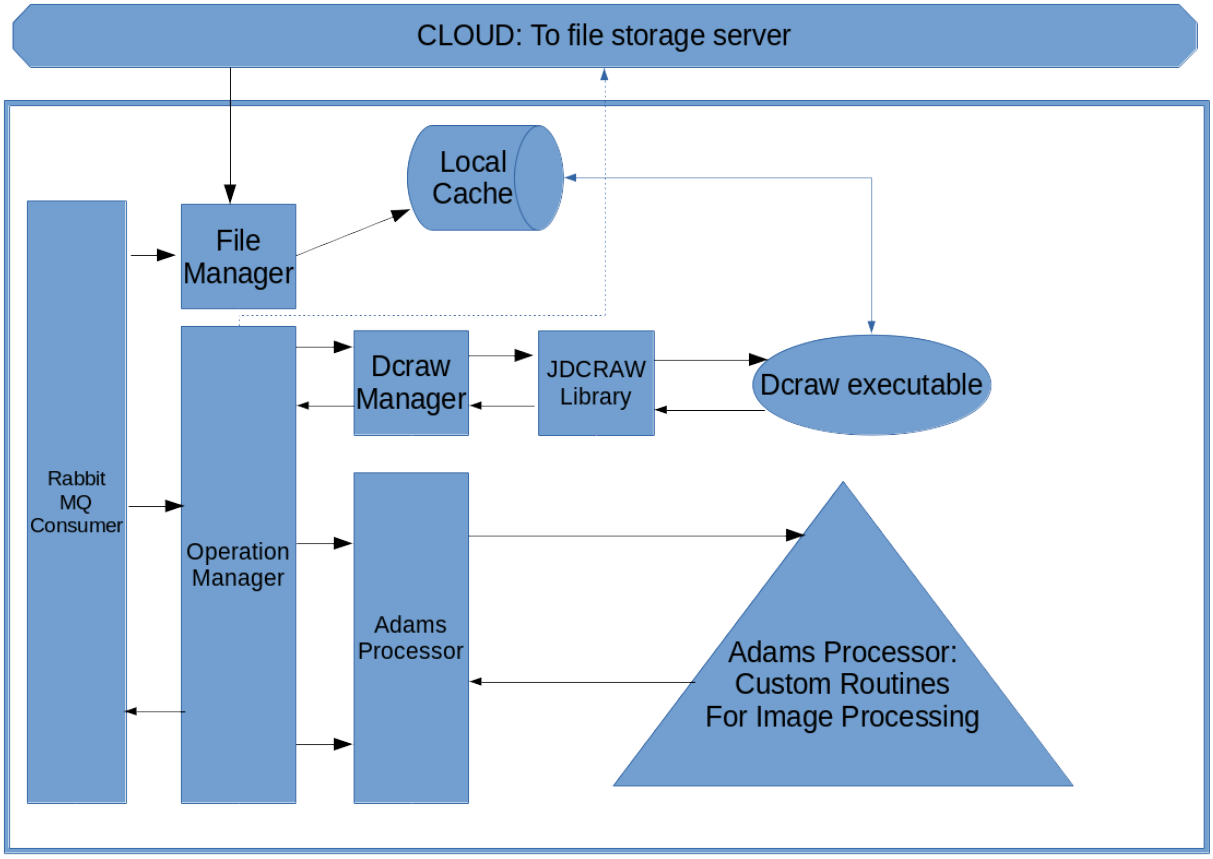

Figure 2: A diagram showing the structure of the render server components. The server will fetch from the message queue as JSON object containing the URL to the RAW file, and the settings to render it with. This will be passed to the file manager, which first checks to see whether the file is stored within the local cache, and if not downloads the file from the file storage server, and stores it in the local cache. From here, the Operation Manager takes over, by using the JSON string to execute the appropriate managers. First, the Dcraw Manager takes the RAW file, via the JDCRAW library, and converts the RAW file to an uncompressed TIFF. The location of this TIFF is added to the JSON string, and this is passed back up to the Operation Manager. Then, the Adams Processor (which contains all the custom written Java code for image processing) does the rest of the image processing, and finally writes a TIFF file to the local cache, which then has the URL stored within JSON data structure. Finally, the TIFF image is loaded, converted to base64, and passed back to the message queue along with other image information.

run, and files that should be shared, to make that container work. Rather than customising an individual image, this customisation can be done by building on existing images, saving time without needing to start again from scratch. (Kent 2014)

F Evaluation

In order to measure the success of the system, a mixture of user based tests and system based metrics will need to be taken.

As this is a software product, the underlying server will need to be constant to avoid biased results. Therefore, for the tests, a computer shall be used as the server with the following specs:

- Processor: Intel i7-7700k at 4.2GHz
- Memory: 16GB DDR4-3000 RAM
- Storage: 1TB WD 7200RPM hard drive
- Operating System: Ubuntu 17.10

The amount of file storage here isn't as important, but as we are using a hard drive, file caching might be slightly slower than if using an SSD.

The system will be deployed using Docker Compose, with the configuration scripts shown in the GitHub repository.

F.1 System-based Metrics

Three different images per camera shall be used within the system to test compatibility with other camera manufacturers. Canon, Nikon, Pentax and Lecia camera RAW formats shall be tested.

In order to measure usability (along with the human based experiment), the system shall be used by several different users, and for each, the time taken to submit a preview, and then get the preview displayed on the screen shall be tested, as this figure gives us an indication in the delay between modifying parameters and having the updated image displayed on screen.

F.2 User-based tests

While metrics can give us an indication into the usability and usefulness of the system, it's also important to test the system with users.

The first test shall be carried out with individual users. This first test shall consist of the following simple steps:

1. Log into a given user account
2. Create a new album
3. Upload a new image to that album
4. Open the image for editing in the editor
5. Set the exposure to a specified value
6. Adjust colour settings
7. Use the built-in automated features
8. Export the fixed image as PNG for download

The users selected for this shall be mixed between experienced photographers (i.e. those who are familiar with using current RAW editing software such as Lightroom and Darktable), and people who haven't used any RAW editing software before. The time taken for the users to carry out the instructions shall be measured, along with their comments on the ease of use of the system through a survey. The

use of the time metric allows us to see how easy the system is to become familiar with and to use, while the survey allow us to compare the user's experience and how they found the software to use.

A sample of 10 people shall be used for the live demo, which while it isn't a very large sample size, any sample size larger than this would make the tests difficult to carry out due to the requirement of external application hosting.

III REFERENCES

References

Coffin, D. (2015), 'Manpage of dcraw'. Accessed: 2018-01-10.
URL: *https://www.cybercom.net/ dcoffin/dcraw/dcraw.1.html*

Coffin, D. (2017), 'Decoding raw photos in linux - dcraw'. Accessed: 2018-01-10.
URL: *https://www.cybercom.net/ dcoffin/dcraw/*

Kent, A. (2014), 'Docker, distributed systems, and why it matters to magneto (part 1)'. Accessed: 2018-01-10.
URL: *https://alankent.me/2014/08/28/docker-distributed-systems-and-why-it-matters-to-magento-part-1/*

Melnikov, A. & Fette, I. (2011), 'The WebSocket Protocol', RFC 6455.
URL: *https://rfc-editor.org/rfc/rfc6455.txt*

Mozilla (2017), 'Xmlhttprequest'. Accessed: 2018-01-09.
URL: *https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest*

Tezer, O. (2014), 'Sqlite vs mysql vs postgresql: A comparison of relational database management systems'. Accessed: 2018-01-11.
URL: *https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems*

Verhoeven, G. J. J. (2010), 'It's all about the format unleashing the power of raw aerial photography', *International Journal of Remote Sensing* **31**(8), 2009–2042.
URL: *https://doi.org/10.1080/01431160902929271*

Zhu, Q., Mai, J. & Shao, L. (2015), 'A fast single image haze removal algorithm using color attenuation prior', *IEEE Transactions on Image Processing* **24**(11), 3522–3533.