Photo Manipulation in the Cloud

Harry Mumford-Turner
Department of Computer Science
University of Bristol
hm16679@bristol.ac.uk

***Abstract*—Presenting a cloud application for photo manipulation hosted on Google App Engine. The application can perform various image manipulation tasks on a large number of photos. The use case design demonstrates these tasks by first downloading four photos from Instagram, resizing these photos and finally composites them together to create a single photo. The application can be run online at https://front-end-dot-ccinstapute.appspot.com**

I. Introduction

Photographers can take thousands of photos at a single photoshoot event. The photos would be stored after the event, wiped from the camera and each photo would usually be retouched. For example, the photographer could want a watermark to be placed on every photo, or to re-size photos to certain dimensions. These tasks could be programmatically performed using a application on a single computer. However, these tasks such as resizing multiple photos all to the same dimension, are *'Embarrassingly Parallel'*, as it requires little effort to split the problem into a smaller number of parallel tasks. Where it is convenient to process each photo at the same time as each photo does not depend on another, therefore reducing the time for the overall task. When the number of photos to process increases, it is possible to process all of these photos on a more powerful machine, however if more photographers wanted to process photos at the same time, simply increasing the hardware of a single machine to cope with this increase, is not a scalable solution with increased costs, increased risk of loss from a single computer failure and difficulty to handle an unstable load. Our application copes with these issues by using a scalable architecture. It processes each photo in parallel on multiple machines in the Cloud hosted on Google App Engine. The application scales horizontally by adding more processing power for the application to account for the increase in users and photos to process.

To demonstrate the architecture design of the application, a basic implementation has been developed, limiting the functionality to only two photo manipulation techniques on four photos downloaded from the popular photo sharing service Instagram [1]. An Instagram user can be identified, four photos from a particular Instagram user are downloaded, resized in half and composited onto each other to create a single photo.

The application has been designed for scale by using several microservices to complete the goal. *'Microservices allow a large application to be decomposed into independent constituent parts, with each part having its own realm of responsibility'* [2]. Each microservice processes work independently from one another and were built in the cloud using Google App Engine, one of the leading providers for Platform as a service (PaaS). This choice was ideal because each microservice can be managed easily by the App Engine and the application setup and configuration was faster. *'Google App Engine is a scalable system which will automatically add more capacity as workloads increase.'* [3].

II. Implementation

The application is written in Python with two frameworks to ease the development of the microservices. Three microservices were constructed to separate the application into different parts, this utilised the Google Cloud Computing benefits by autoscaling, load balancing, caching with Memcache and managing instances for each service. Therefore, when the load increased for a particular service, that can be scaled accordingly. Google App Engine APIs were used to create Task Queues for jobs and manipulate photos. The Google App Engine Cloud Datastore and Storage were used to store information about a job and to store downloaded photos.

The application uses Flask [4] to aid with Python development and WebApp2 [5] a web framework that is compatible with the Google App Engine and provides easy routing. To create a microservice, an app.yaml file specifying details about the service and what libraries it depends on is required. The interface between the client and the microservices is where WebApp2 framework comes into play, it is a (Web Server Gateway Interface) WSGI and results in an easy transfer of information from web requests.

The program is made up with the following structure. A Front End service, a Download Photos service and a Manipulate Photos service. These three microservices work independently from one another. They are created using a simple python folder structure. A *templates* directory for HTML templates, a *static* directory for all static files, such as CSS files, a *library* folder accompanied with requirements.txt to keep 3rd party libraries and finally the three microservices each identified using the format, *microservice-name.py* and *microservice-name.yaml*.

Users can start new jobs from the front-end service by using an Instagram username and submitting a form. This process starts a new job by creating a new Entity in cloud storage and adds the job to a queue which is later processed by the Download Photos microservice. Google has several storage options on offer, a Google Cloud Datastore [7], that stores data types for property values, the most efficient solution to store simple variables for information about a job. This data is then accessed throughout the application using a unique identification for a job.

Google Cloud Storage [6] is the second cloud storage option used. The Download Photos microservice processes the queue and writes the downloaded photos to a bucket. This bucket stores the Photos that are later processed by the Manipulate Photo microservice.

A. Website Design

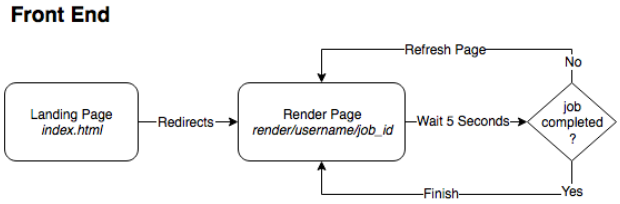

Fig. 1. User perspective of the Front End

In the above figure 1 when a user visits the landing page, types in a username, then submits the form. A HTTP web request is made to a WebApp2 function to start a new job, */start-job*, after a new job has been successfully created, the user is redirected to the */render* page loaded with information, such as a unique identified about the newly created job. An simple javascript refresh method was implemented to check if the job has been completed, if it has not yet finished, it refreshes the page. When the */render/username/job_id* route is accessed, the job is loaded from Google Cloud Datastore, using the unique identifier, the *job_id*, to check if it has completed. While this process is happening the front-end service is not blocked and displays a loading icon to the user.

B. Front End Microservice

The WebApp2 function start-job in the front-end service creates a new Entity with the given username, then creates a unique key for that entity and uses that for the job identifier. This identified is sent to a Push Task Queue [8] which is used to queue up tasks, such as download these photos for this user, or manipulate this photo. After the job has been queued the user is redirected to the */render/username/job_id* url and the *render.html* template is rendered. The other microservices will process the jobs in the Task Queue, then populate the Entity with the final result and change a completed flag in the Entity. This means that multiple jobs can start at the same time and makes the application scalable.

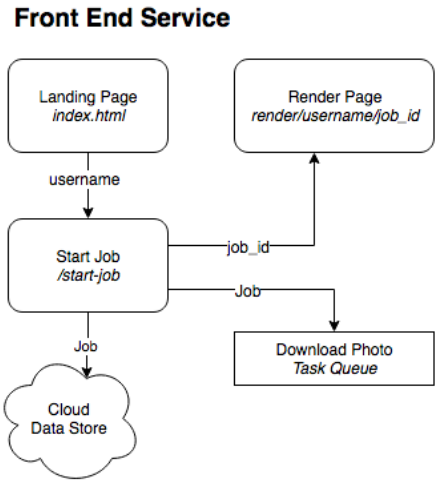

Fig. 2. Front-End Microservice overview

C. Download Photos Microservice

The Download Photos microservice */process-work* function gets called from the Download Photo Task Queue. It uses the informationfrom the Task in the Queue, such as a URL to a photo, to download the photos from a photo service, using information about the job, in this case the Instagram username to filter photos. Once one photo has been downloaded the photo data is added to another Task Queue, this time for Photo Manipulation, the data from the job is also added, so the next microservice knows how to alter the photo. Configuration limits for the number of photos to download at one time, and the amount of photos to add to the queue are added, which scale easily, due to the nature of microservices.

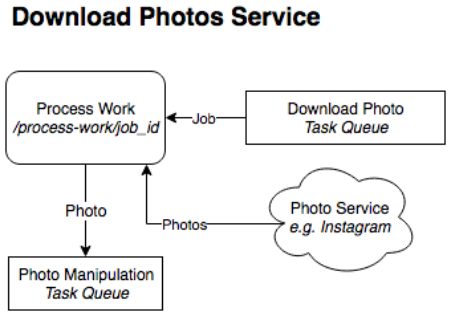

Fig. 3. Download Photos Microservice overview

D. Manipulate Photo Microservice

The */build-image* function is called to process a photo in the Photo Manipulation Task Queue. The information is extracted from the Task and the photo is manipulated accordingly. In this example, the photo is resized to a half of its original size. Afterwards the photo is saved in a Cloud Storage Bucket, its name and a completed flag are appended into an array into the Entity based on the *job_id*.

The functionality of this microservice can easily be extended into multiple image manipulation techniques, however, third party image processing libraries, such as PIL [9] are required.

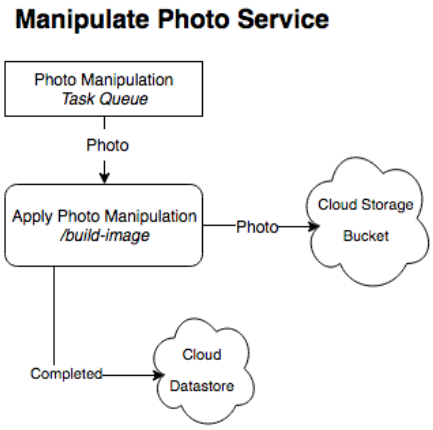

Fig. 4. Manipulate Photo Microservice overview

E. Application Architecture

The overall architecture could be achieved in a single service, however, this structure copes with sudden increase loads in different parts of the system.

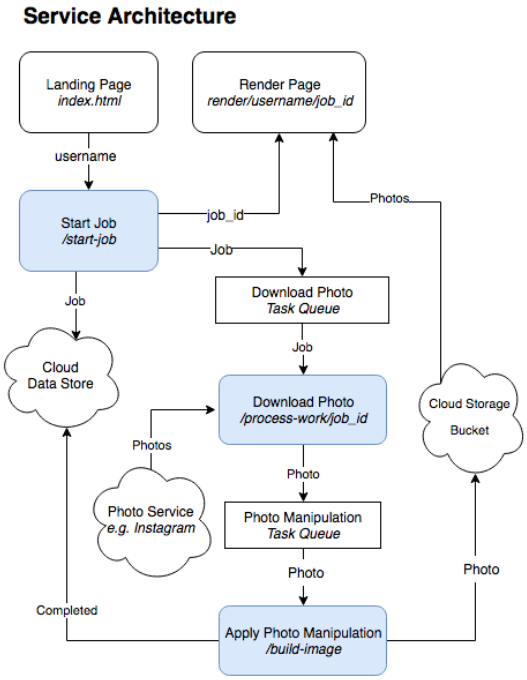

Fig. 5. Application Architecture

III. Scalability Issues

The initial choice to use Google App Engine over Amazon Web Services [10], proved effective when scaling. Amazon EC2 IaaS option (Amazon Elastic Compute Cloud) [11] does not scale automatically as it is simply a virtual server. However, EC2 instances can be scaled using Auto Scaling Elastic Load balancing, however a large amount of administration work is required. With Google App Engine, this process happens automatically, so is not the case. This PaaS offering means users saves time in setting up middleware, however flexibility is reduced because they only have access to the services exposed by the particular middleware that is combined in the platform.

The chosen cloud storage options scale well compared with a traditional SQL database. Google App Engine Cloud Datastore offers a managed NoSQL database that scales automatically. However, with a NoSQL database there are limitations with queries and transcations. *'If you update an entity group too rapidly then your Datastore writes will have higher latency, timeouts, and other types of error. This is known as contention.'* [3].

Google App Engine provides lots of configuration about how the automatic scaling behaves and has not be explored well enough. For example, particular microservices could have their settings adjusted for when their instances are served or how the application handles delays in requests.

Google provides a caching system called MemCache to speed up operations that frequently occur. This process is automatic and will reduce the resource costs for running the application. However, because the Google App Engine tries to have little configuration to get started, tweaking the configuration if the application has a regular load would be beneficial for resource costs.

IV. Conclusion

Building an application using microservices is beneficial for scaling, and is cost effective when the load on the different microservices is unstable. Instapute demonstrates how multiple photo downloads and manipulation can work in the cloud as a scalable solution.

References

[1] Instagram, Developer Documentation, 2016, https://www.instagram.com/developer/
[2] Google, Microservices Architecture on Google App Engine https://cloud.google.com/appengine/docs/go/microservices-on-app-engine
[3] Google, Designing an App for Scale, December 2013 https://cloud.google.com/appengine/articles/scalability
[4] Flask, Python Micro Web Framework, April 2010 http://www.flask.pocoo.org
[5] WebApp2, Python Web Framework for Google App Engine https://webapp2.readthedocs.io/en/latest/
[6] Google, Entity Property Reference https://cloud.google.com/appengine/docs/python/ndb/entity-property-reference
[7] Google, Entity Property Reference https://cloud.google.com/appengine/docs/python/ndb/entity-property-reference
[8] Google, Push Task Queues https://cloud.google.com/appengine/docs/python/taskqueue/push/
[9] PIL, Secret Labs AB, Python Imaging Library, December 1995 http://www.pythonware.com/products/pil/
[10] Amazon, Web Hosting Amazon Web Services, 2016, https://aws.amazon.com/websites
[11] Amazon, Amazon Elastic Compute Cloud, 2016, https://aws.amazon.com/ec2/