# Image Evolution Service
## Creating scalable web applications using cloud services

William Strickland

University of Central Florida, Orlando, Fl, 32817 USA

**Abstract**

This paper examines the use of cloud based services to build a application that while it does not have apparent commercial value does exhibit many of the same characteristics of resent cloud startups and several features that make it well suited to deployment on Platform-as-a-Service. This paper focusing on the design of a service that accepts user submitted images and approximates them using evolutionary programming techniques to output a stylized image with an conceptually similar to Rotoscoping of photographs or live action film to provide an interesting aesthetic. Design decisions are focused around turning the simple application into a service that would be reliable and scale well on cloud infrastructure.

## I. Introduction

Rotoscoping is an animation techniques originally developed in 1917 to provide more realistic movement to animated films [4]. Since then it has been used extensively to add realism to animated films as well as provide surrealism to photographs, animation and live action films. A famous example of this is the use of a computer assisted rotoscoping technique for production the 2006 movie “A Scanner Darkly” [5]. The primary operation in rotoscoping is to trace, traditional by hand, the major lines in an image and then make flat color. The success of this technique, whether in being realistic or surrealistic, depends on the skill of the artist doing the trace.

Rotoscoping provides a interesting aesthetic, but what would it look like if we took the artist out of the process and used a algorithm instead. Some attempts have been made to do with by using line finding algorithms, but the results have been mixed. At some point the question was asked if a genetic algorithm could do this in an interesting way and what would that result look like. If the problem is reformulated to define a set large (important) shapes of similar color instead of a set of lines that separate regions of similar color we can start to design a algorithm that could approximate the image.

## II. Related Work

The first related work in this area was that of Roger Alsing to evolve an image of the Mona Lisa. Alsing set the groundwork for future work by defining the problem as a set of polygons of a given color and transparency that can be compared pixel by pixel against the original image. In his implementation, Alsing used a technique similar to simulated annealing to move from a random initial image to one closer and closer to the input image. The current best image is used to create is compared against a mutation of the current image and the one with the better fitness is kept and the process is repeated. While others have commented that this simply simulated annealing or some other hill climbing algorithm, Alsing maintain this is an evolutionary computation algorithm with a population of two . This program was written in C# and released under GNU Lesser GPL license in December of 2012 [1].

Following Alsing's work, others attempted to port this problem to JavaScript. This was done largely as a technical demonstration of the computational abilities of HTLM5 and JavaScript in modern browsers [2][3]. Because of the graphical nature of the problem and difficulty of computation it made sense to use this a show case for the new capabilities of JavaScript and HTML5 for graphics. AlteredQualia presented an direct adaptation of Alsing's design to JavaScript but added some more options in how mutation is done as well as a great deal of research into the efficacy of the algorithm itself [3]. Later, Jacob Seidelin followed Alsing's work with a true genetic algorithm implementation

in JavaScript in January of 2009. Jacobs design focused on quickly getting to a moderate level of closeness to the original image quickly. In his design and testing it was not confirmed whether Seidelin's design would eventually allow for the level or accuracy demonstrated by Alsing [2].

It was found in all of these applications that compute time for even small images to become recognizable would be on the order of 20-30 minutes on a typical personal computer systems. High detail image evolutions were found to take days on all of the previous implementations. This would made the application prohibitive on mobile platforms and tedious for users that would have to leave the application running on their own machine for long periods of time.

## III. Application Architecture

The goal of this project was to build this concept into a service provided to users. Inspiration was derived partially from how the Instagram service provides artistic filters of user uploaded images as well as a social network around these images [6]. This would require a web application be built around the image evolution code. In fitting with the design goals of this project, the web application would be designed in a scalable and reliable way such that the failure of individual systems would not result in the loss of data or service.

The first design challenge to creating this service would be to move the computation and storage off the client machine and the service infrastructure. This would enable the application to be run on many compute limited environment and eliminate the frustration on users to leave their system running to process these images. The service could operate in request and forget way. Users could request and image and checkup later to see the progress or even be notified later when the processing had completed. This would require a great deal of processing power from the be available to the service and that computing power would need to scale with user requests.

The Amazon Web Services (AWS) cloud was chosen as the hosting platform for this application due to the variety of cost effective services provided to enable the creation of scalable web applications. To provide the computational capacity to run the web application and perform the image evolutions. Components of the AWS platform were chosen to solve individual challenges of this application. Components were chosen in such a way as to provide maximum scalability while minimizing costs and IT overhead.

Computational resources for the application are provided by AWS Elastic Beanstalk. Elastic Beanstalk provides Platform-as-a-Service capabilities for many established web programming frameworks. In particular, the Apache Tomcat is well supported and was chosen as the application framework due to familiarity and example projects in the many key areas of the required scope [7].

Elastic Beanstalk works by aggregating and simplifying many other AWS products including EC2 for virtual servers, Elastic Load Balancer and auto-scaling to provide elasticity. Elastic Beanstalk provides a managed application platform as Amazon maintains the operating system and platform libraries so long as standard images are used. In this way the developer is only responsible for the code deployed in a standard means to the server. In the case of Tomcat, as Elastic Beanstalk brings up new servers, they fetch and start the Java Web Application Repository (WAR) provided to Elastic Beanstalk from Amazon S3 [7].

Elastic Beanstalk application servers (instances) are created and destroyed without warning. Data stored on an instance is volatile so application data must be persisted a more long term storage service. Also, servers are effectively stand alone so to coordinate activity between servers a centralized storage service is helpful.

For this application, AWS Simple Storage Service (S3) is used to provide long term storage of original images and the evolution results. S3 provides high reliability and availability for data stored [8]. It is easily integrated in to Java applications using libraries provided freely by Amazon [16]. The S3 service is completely web based and provides many options for developers on how files can be retrieved and with what level of security [8].

While S3 is very cost effective and provides good performance and scalability for web applications, it does not provide effective storage for small, but frequently searched data (such as image metadata). Also, to coordinate the action of

the many application servers launched by Elastic Beanstalk, a system for communication and building consensus is needed. This role is usually done by a central database or database cluster with applications. Unfortunately, scaling traditional SQL databases is costly and limited. In fitting with the design goals, AWS SimpleDB was chosen as the database solution for this application. SimpleDB is simple to setup and provides all the primitive operations needed to efficiently communicate and achieve consensus between the application servers [8]. SimpleDB's capabilities are very limited compared to other NoSQL options commonly used on the AWS platform, such as AWS DynamoDB or using EC2 to create MongoDB instances. However, it's capabilities and target uses cases exactly match the requirements of this application. SimpleDB acts as a key-value storage system similar to S3. However, instead of files stored for each key, there is a set of attributes with a set of values for each attribute [8]. This design is similar to Google's BigTable [12]. SimpleDB provides conditional update statements which allows consensus protocols to be built between servers though the implementation of specifics require careful consideration to prevent the host of issues that plague parallel computation. The only down side to SimpleDB is its relatively complicated support in the amazon provided libraries [16]. These libraries can be difficult to work with at time, but once a basic design pattern is reached it can be replicated for all interactions with SimpleDB (forming another abstraction layer on top of the library.

While SimpleDB has the capability to also serve large pieces of information (such as files) it is less suited to this task compared to S3 [11]. Similarly, while job queuing and coordination could be done using SimpleDB alone, it is not optimal.

For distributing image evolution jobs between servers another AWS product was chosen. AWS Queuing Service (SQS) provides a robust and cost effective way to pass messages between a large number of distributed servers. The service is well designed to scale and provides an effect API which is easily integrated using the amazon provided libraries [10][16].

The requisite code for image evolution was built deep within the scalable web application. This code was designed so that it could be integrated into a client application for initial development and testing and the final web application. The Java AWT libraries were used to implement the graphics portions of the code due to their simplicity to use and good support for input and output configurations. Both genetic and hill climbing algorithms were designed based on prior works my Alsing, Seidelin and AlteredQualia. The evolution core code was designed to be very modular and provide access to all combinations of mutation method and optimization implemented. The operation of this core is governed by an input object passed to the runnable main class. This allows a driver application to configure an evolution (typically from an SQS message) and then run that evolution as a thread. The driver can then move on and wait for the thread to finish. This design is well suited to integrating into the back-end of a web application.

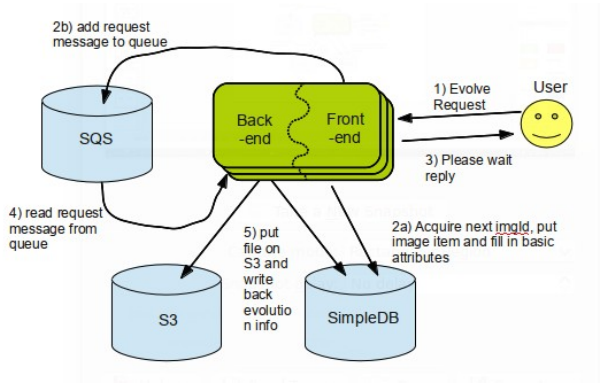


*Figure 1. diagram of user request workflow showing key interactions.*

## IV. Implementation Details

This service was implemented in the Java programming language version 1.6 using the Apache Tomcat 7 framework. Development was done on a Linux desktop configured with similar Java and Tomcat versions using the Eclipse JEE IDE (4.2 Juno) and AWS plug-in. Setup, Management and inspection activities for S3, SQS, SimpleDB and Elastic Beanstalk services were done directly from the Eclipse by means of the AWS plug-in.

Rather than implementing user management into the application it was decided that it would be more scalable and provide better user experience to federate user accounts using OpenId. Using the OpenId protocol. The OpenId protocol allows users to use accounts with other service providers such as

Google for this site. The OpenId provider selected by the user is provides this application with a unique user identification and requested user information such as the user email and user nickname [13]. The OpenId4Java library was used to provide OpenId integration for this service [15].

For the implementation of this service the image evolution core process was integrated into the WAR of the web front end using a scheduled process that polls SQS for evolution jobs. This was decided to the better design compared to having two separate applications for the web application and evolution portions. Having the evolution portion integrated into the web application makes the environment more homogeneous by having one type server running. It also allows the minimum number of servers to be one instead of two. This was advantageous during testing due to the light load, but would not be an advantage as the app scales.

One disadvantage to having the application structured this was is the apposing metrics needed for the auto-scaling functionality of Elastic Beanstalk. Auto-scaling is capable of monitoring many metrics to determine if it should scale up or scale down the environment. Arguably the best metric for the web application is request time. However, the best metric for evolutions is CPU utilization. It was decided that CPU utilization an acceptable metric for the web front end as it was observed that the primary slow down of requests was evolutions taking place on the box. The thresholds for scaling on the application were configured at 75% and 25% average CPU utilization for scale up and scale down respectively. The maximum instances was left at the default and conservative count of 4 along with the default minimum of 1.

Integration of the web and evolution portions was accomplished by setting a servlet context listener in which the contextInitialized method will be called on server start allowing a thread to be spawned to poll and start evolutions. Each server has a set number of evolution 'slots' in which it could pull a job from the SQS queue. This number was hard-coded into the application. The evolution process starts new threads to perform the evolution computations and iterates the slots to looking for completed jobs it can write back to S3 and SimpleDB. When a slot is emptied it will begin polling SQS looking for a new job to fill that slot. After approximately a minute it will move on to the next slot and either skip over or attempt to write back and fill that slot.

SimpleDB is used to manage three types of information in the application: users, sessions and images. Each is given its own domain (table) in SimpleDB.

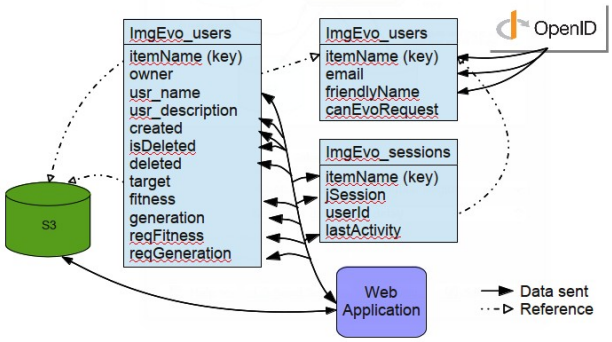


*Figure 2. Dataflow diagram of application showing the SimpleDB domains as well as the source for the data contained in them.*

Due to difficulties imposed by the limitations of Elastic Beanstalk to create unique sessions and share them among Tomcat servers it was necessary to create a common session domain and share it among the servers. The session identifier is given to the users as an opaque session cookie and all requests in the application are checked against the user permissions before being performed. The session identifiers are randomly generated and reserved in SimpleDB. Conditional updates are used to make sure that exactly one user and session gets a particular session id. Should reserving a session id fail, the task will repeat with a new randomly chosen id.

The session ids are fairly long randomly chosen using a cryptographically strong pseudo-random number generator. This means that collisions are rare and difficult for attackers to predict. HTTPS and Transport Layer Security are used to secure the identify of users and their requests. Insecure HTTP connections to the application are not allowed as this could allow users to mistakenly enable attackers to spoof their session and take action using their credentials.

The user domain in SimpleDB is used store user information provided by OpenId as well as the permissions given to a users. The permission of key importance governs whether or not a user can

request evolutions. This prevents just any user that signs in using OpenId to monopolize resources during the testing phase.

The most important SimpleDB domain to this application is without question the images domain. This domain stores the metadata for images requested and evolved by the application. It serves as the synchronization and consensus point for the application servers and is the focal point of the web application. In a process similar to session id generation, image id are randomly reserved in the database using conditional updates guaranteeing that exactly one server and image gets a particular image id. This time however, the pseudo-random number generation need not be cryptographically strong so the performance penalty of strong random numbers is avoided here.

After evolution computation is complete, the fitness and number of generation is written back to the entry for the matching image in this domain. Also, ownership information that determines which images a user can see is recorded in this table by the user id key matching the user table.

SQS messages contain a JSON description of the job to be done. The current implementation provides only a basic amount of information about the job and requires that most of the evolution parameters be hard-coded. This could be easily extended and due to the flexibility of the JSON in Java libraries used to encode the SQS messages [14]. With a more complex user interface and message parsing the fully range of options implementation in the evolution core could be exposed to the user.

Images stored in S3 are protected at all times. Permissions to read and write to S3 are limited to the key pair used by the application. However, Images must be retrieved from S3 and displayed to the user. To accomplish this the application creates signed URLs on page request using it's key pair which allow the user's browser to retrieve the image file for a few seconds. It is felt that this is an appropriate, but not unbreakable level of security to offer. The user has the ability (for a short time) to pull images they own out of S3 where they can be safe guarded by the application. This is an ability that the user should have if they chose to utilize it and as this is not an obvious thing to see how to do it can be assumed that the user would not do this inadvertendedly.

The Elastic Beanstalk configuration was setup to use EC2 Micro instances for the application servers because of cost effectiveness of these servers. A side benefit is that their lack of computational power means that it requires fewer evolution jobs to cause them to scale. However, because these servers have a single virtual core where configured to have two evolution job slot. This means at most 2 jobs could be processed at once on a server. So if the number of servers is 8 the maximum number of jobs being actively computed is 16. Fortunately, many more jobs can live in the queue.

## V. Results

The initial phases of design and programming focused on creation and testing of the evolution core. After implementing and testing most of the design features of the previous works cited as well as some original ideas it was determined that my Java implementation achieved similar performance and results to the reference implementations and the best algorithm configuration overall was the hill climber with mutations limited to one one parameter of one polygon per round.

Optimal polygon count and number of vertices per polygon varied from image to image but it was found that 100 6-vertex polygons was good for most of the small test images. It was also observed that the most important factor in performance was the size of the target image in pixels. This is due to the very expensive fitness calculation performed on each new candidate image. The computation effort for mutation of candidates, picking new generations and rendering a image from a set of polygons.

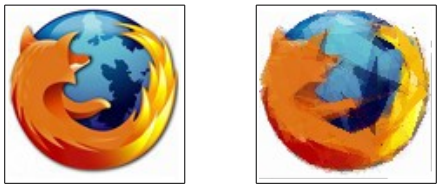

*Figure 3. Left, Mozilla Firefox logo used as target image. Right, evolution result after 16 hours of computation with hill climber algorithm; best of 1,206,408 candidates tried, 93.17% fitness.*

The evolution core continued to perform as expected when placed inside the web application and performance was not noticeably hurt by making

calls to S3 and SimpleDB to fetch target images and write back image data. However, the use of EC2 Micro instances did cause some problems. The single core of these instance would instantly go from less than 10% to 100% utilization at the start of a single job. This resulted in difficulties with the auto-scaling configuration with a small number of servers. The average utilization would cycle high and low quickly and prevent scaling. With a large number of servers this pattern would be smoothed.

It was also noticed that web application response time suffered strongly when the request handling server was running a job. This could be remedied by changing the thread priority on the evolution jobs to give web traffic handling higher priority and thus improve web experience.

The SQS read visibility timeout was not large enough in many cases and that lead to some images being processed on multiple servers and the same time. Luckily, the application was prepared for this and the conditional update logic implemented made sure that the higher fitness image was kept.

The application was able to distribute evolution jobs and web traffic among all servers with no obvious signs of hot-spotting or concurrency issues.

## VI. Conclusions

This project was a partial success in terms of implementing a reliable, scalable web service that allows users have their images evolved.

The implementation achieved shows that most of the choices made in architecture perform well and allow for operation in a large scale distributed system. This conclusion carries little weight however due to the small scale and limited testing that took place. Additionally, there were some key features originally planned that would have greatly increased the robustness and scalability of the application that were not incorporated due to time constraints.

The most important of these is allowing evolution jobs to take as input a previous set of candidates (exported as DNA to S3). This would allow the application to write small regular chunks back to S3 and make them available to the application by SimpleDB update. This would resolve the issue of SQS read timeouts and with it largely solve the problem of duplicate work being done. This would also allow jobs to be started more quickly and provide the user better feedback on the status of their evolutions.

An important conclusion for this project was that utilizing AWS cloud services was helpful despite the limitations they brought. The integration of cloud services was one of the easier portions of the development effort. This was probably largely due to specifically picking cloud services that were well suited and could be easily adapted to the application needs.

## VII. Future Work

There are many functionalities that were originally planned and in some cases even designed that had to be dropped due to time constraints on development. Some of the focus on application performance and robustness while others contribute to user interface. Some even take the app closer to the socially integrated portal concept that was originally envisioned.

First and foremost, the addition of doing evolutions in stages and writing back incremental results while putting a new queue record into SQS for some other server (or potentially the same one) is critical for future research using this application. The advantages this provides in application checkpoints for reliability, better job distribution, better response time and status updates to users is of great value. The biggest challenges to implementing this are DNA export/import and coordinating SQS message delete and creation such that it is reliable in the event of failures.

Two other enhancements that would help to a lesser extent with performance are dynamic job slot sizing and revised thread priorities. Allowing the evolution service to determine how many evolution threads it can effectively run would take some of the guess work out of migrating the code between different instance types. This is however difficult to do and may not be worth doing considering that this would be rare. Thread priority revision would be an easy addition and would definitely help with web request issues when jobs are running.

While not directly related to the scalability of the application, the user interface could use improvements in many areas. First, a greater selection of the implemented options in the evolution core belong on the evolution request page. Choice is a core part of the application, and it

is currently very limited. Another issues is the lack of input sanitation which could cause issues if the application was released to the public and became a target for hackers. Also, using AJAX to display updates to the page without requiring the user to manually reload would improve user experience.

Finally, several key features are missing that would demonstrate the original concept and better demonstrate the distributed nature of the application. Users should receive notifications, if they chose, when their images are done. This would help establish the application as the request and come back later model it was originally designed for. Without this feature it forces users to reload pages repeatedly to see if their image is done. This functionality could be implemented easily with AWS Simple Notification Service. Another missing aspect is the social integration. The original intent was a portal where users could not only have their images evolved, but also share the results with friends and the world at large. To incorporate sharing with friends, the ability to discover other users and build friends lists is required. Additionally, images would have to be flagged with multiple users that have been shared an image. To share an image to the public it would make the S3 service backing the application public for some images. This would allow images to be posted around the Internet. In the same way the services like Flickr.

## VIII. Appendix

The source code and documentation for this project is provide on google code under MIT license. You can find the google code project at http://code.google.com/p/cop6938-image-evolution/

The following are selected images of the application running to provide context to the discussion.

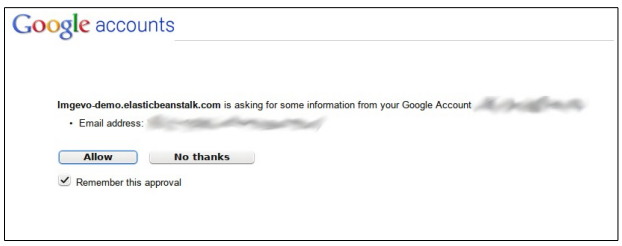


*Figure 4. OpenId federation, user prompted for permission to provide application user information from Google.*

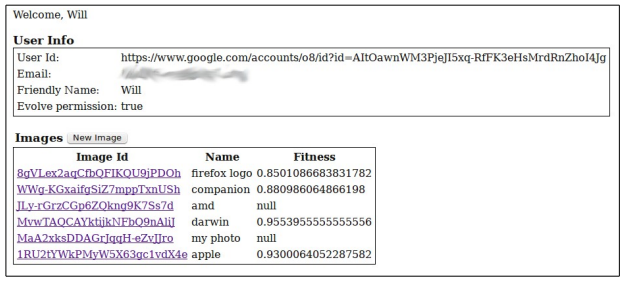


*Figure 5. Application dashboard.*


*Figure 4. Example of completed evolution with 'low' fitness requirement.*

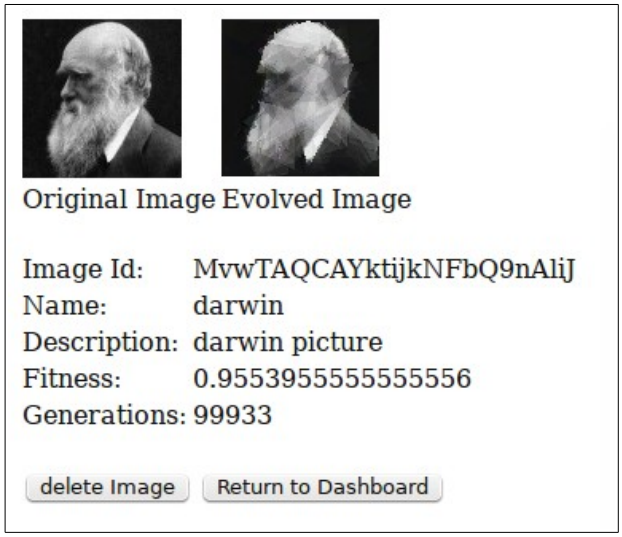


*Figure 4. Example of completed evolution with high fitness requirement.*

## IX. References

[1] Alsing, R. (2008, December 07). *Genetic programming: Evolution of mona lisa*. Retrieved from http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa/

[2] Seidelin, J. (2009, January 09). *Genetic algorithms, mona lisa and javascript canvas*. Retrieved from http://blog.nihilogic.dk/2009/01/genetic-mona-lisa.html

[3] AlteredQualia. (2012, August 18). *Image evolution*. Retrieved from http://alteredqualia.com/visualization/evolve/

[4] US patent 1242674, Max Fleischer, "Method of producing moving-picture cartoons", issued 1917-10-09

[5] Levine, R. (2006, July 07). A long way from disney. *Los Angeles Times*. Retrieved from http://articles.latimes.com/2006/jul/07/entertainment/et-rotoscope7

[6] Frommer, D. (2011, November 01). *Here's how to use instagram*. Retrieved from http://www.businessinsider.com/instagram-2010-11?op=1

[7] *Aws elastic beanstalk overview*. (2012). Retrieved from http://aws.amazon.com/elasticbeanstalk/

[8] *Aws simple storage service overview*. (2012). Retrieved from http://aws.amazon.com/s3/

[9] *Aws simpledb overview. (2012). Retrieved from http://aws.amazon.com/simpledb/*

[10] *Aws simple queuing service overview*. (2012). Retrieved from http://aws.amazon.com/sqs/

[11] *Aws simpledb faq*. (2012). Retrieved from http://aws.amazon.com/simpledb/faqs/

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (OSDI '06), Vol. 7. USENIX Association, Berkeley, CA, USA, 15-15.

[13] *Benefits of openid*. (05, December 20). Retrieved from http://openid.net/get-an-openid/individuals/

[14] *Openid4java - openid 2.0 java libraries*. (2012, December 05). Retrieved from http://code.google.com/p/openid4java/

[15] *Json in java*. (2012, December 05). Retrieved from http://www.json.org/java/

[16] *Aws sdk for java*. (2012, December 05). Retrieved from http://aws.amazon.com/sdkforjava/