Foundry:

A Cloud-Based Web Application for Transforming User-Generated Content

by

Conor Smith, B.A. (Mod.)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2009

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Conor Smith

September 9, 2009

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Conor Smith

September 9, 2009

Acknowledgments

Firstly I would like to thank my supervisor Stephen Farrell for his guidance throughout my work on this dissertation.

I would also like to thank my friend and employer Robert Donohoe for his support and lenience during my work on this dissertation. Thanks to my friend and fellow NDS classmate Colin Lyons for being a sounding board for my ideas and problems throughout the year. Thanks as well to Cathal Horan and all attendees of his 21st birthday party for providing test images for my application.

Finally, on a somewhat less serious note, I would like to thank Ballygowan water for providing ample hydration throughout my work on this dissertation.

Conor Smith

University of Dublin, Trinity College
September 2009

Foundry:

A Cloud-Based Web Application for Transforming User-Generated Content

Conor Smith, M.Sc.
University of Dublin, Trinity College, 2009

Supervisor: Stephen Farrell

User generated content (UGC) has been a growing trend on the web over the last few years. UGC is created, modified and consumed by the users of web applications. Cloud computing is another area that is quickly developing. By using a cloud computing platform applications can be devised that perform massive varieties of functions on UGC and store this newly generated content, just in case users are interested in the results.

This dissertation aims to implement a large-scale web application on an infrastructure-as-a-service cloud computing platform that allows users to upload their own images, that performs transforms on these images based on user specifications, that stores all of its content in the cloud and that is able to scale when presented with additional users and data.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **ix**

List of Figures **x**

Chapter 1 Introduction **1**

Chapter 2 State-of-the-Art **3**

2.1 Large-Scale Web Applications 3
2.1.1 Software Architecture 4
2.1.2 Hardware 7
2.1.3 Scaling 9
2.2 Cloud Computing 10
2.2.1 Everything as a Service 10
2.2.2 Commercial Cloud Computing Services 12
2.3 Amazon Web Services 14
2.3.1 Amazon Elastic Compute Cloud (EC2) 14
2.3.2 Amazon Simple Storage Service (S3) 18
2.3.3 Amazon SimpleDB 19
2.3.4 Amazon Simple Queue Service (SQS) 21
2.3.5 AWS Security 22
2.4 User-Generated Content 23
2.4.1 Content Processing 23
2.4.2 ImageMagick 24
2.5 XML and Schema Languages 24
2.5.1 XML Schema 25

Chapter 3 Design **27**
3.1 Requirements . . . 27
3.1.1 Functional Requirements . . . 27
3.1.2 Non-Functional Requirements . . . 28
3.2 Use Cases . . . 28
3.2.1 Account-Related Use Cases . . . 28
3.2.2 Content-Related Use Cases . . . 30
3.3 Architecture . . . 31
3.3.1 Functional Architecture . . . 31
3.3.2 Functional Architecture Messages . . . 31
3.3.3 Functional Architecture Components . . . 33
3.3.4 Technical Architecture . . . 35

Chapter 4 Implementation **39**
4.1 Storage . . . 39
4.1.1 APIs . . . 39
4.1.2 S3 . . . 39
4.1.3 SimpleDB . . . 41
4.2 The Web Servers . . . 42
4.2.1 Initial Implementation . . . 42
4.2.2 AMI . . . 42
4.2.3 Configuration Files . . . 43
4.2.4 Creating Jobs . . . 45
4.2.5 The User Interface . . . 46
4.3 The Processing Servers . . . 49
4.3.1 AMI . . . 50
4.3.2 Processing Jobs . . . 50
4.4 Launching the Application . . . 51
4.4.1 Load Balancer . . . 52
4.4.2 Launch Configurations . . . 52
4.4.3 Auto Scaling Groups . . . 52
4.4.4 Triggers . . . 52

Chapter 5 Evaluation **53**
5.1 Testing . . . 53
5.2 Scalability Analysis . . . 55
5.2.1 Usage Increases . . . 55

5.2.2 Data Increases . . . 56
5.2.3 Maintainability and Functional Scalability . . . 57

Chapter 6 Conclusion **58**
6.1 Foundry . . . 58
6.2 Future Work . . . 59

Appendix A Example XML Schema **60**

Bibliography **63**

List of Tables

5.1 Processing Server Test 1 54
5.2 Processing Server Test 2 54
5.3 Processing Server Test 3 55

List of Figures

2.1 Layered Software Architecture for Large-Scale Web Applications 4
2.2 Cloud Computing Services Stack 11
2.3 Elastic Load Balancing Conceptual Architecture 16
2.4 Auto Scaling Example 17

3.1 Use Case Diagram 29
3.2 Functional Architecture 32
3.3 Technical Architecture 35
3.4 Network Architecture of AWS Components 36
3.5 Web Server Software Stack 37

4.1 Registration and Login Page 46
4.2 Front page when logged in 47
4.3 An example of a gallery page 48
4.4 An example of a transform gallery page 49
4.5 The AWS Management Console 51

Chapter 1

Introduction

One of the cornerstones of *Web 2.0* is user-generated content (UGC). Whether its text, photos or videos website users across the globe have been uploading their own work to various sites. In fact, the amount of UGC being created online everyday is more than four times that of traditional content being created daily [66].

Meanwhile, another trend of today's web is that of cloud computing. Vast amounts of content, web sites and entire applications are being put "on the cloud". Companies such as Google have already heavily integrated their businesses into the cloud. Google offers cloud-based applications such as Gmail and Google Docs, which run from and store users' information entirely within Google's cloud.

The motivation behind this dissertation was to create an application based within the cloud that would take advantage of the scalability that cloud computing platforms are purported to provide. The application was to take advantage of the UGC trend by providing users with some application where they upload their own content, which is then processed in a variety of ways just in case the user is interested in the outcome.

The goals of this project are to design and implement a web application capable of scaling to a large userbase (in the order of millions), running on a cloud computing platform, accepting user-generated content and performing some process on this content, and evaluating the application to determine how much load it can deal with and how maintainable it is.

The structure of the remainder of this paper is as follows:

- Chapter 2 will discuss the state-of-the-art and will provide background into the various areas and technologies discussed and used in this paper.
- Chapter 3 outlines the design of the web application that has been created, including the requirements for the application as well as the functional and technical architectures of it.

- Chapter 4 will describe how the various components of the application were implemented.
- Chapter 5 presents the evaluation of the application, including the tests run to determine the application's scalability.
- Chapter 6 concludes the dissertation, summarising the application and how it fulfilled the project's goals and discussing future work that could be done on this dissertation.

Chapter 2

State-of-the-Art

This chapter presents background on some of the areas discussed and technologies utilised in this dissertation. Firstly, large-scale web applications are discussed along with the concept of scalability. Following that, some background on cloud computing is presented, including information on a number of commercial cloud computing services. This is followed by a more in-depth look at Amazon Web Services, the cloud computing suite run by Amazon and used as the platform for the application being implemented. After this the concept of user-generated content and content processing is discussed. The chapter concludes with some background on XML and XML Schema.

2.1 Large-Scale Web Applications

Today's web is filled with browser-based applications that are used regularly by millions of users. Some applications have to deal with billions of server requests on a daily basis and these numbers keep growing. Obviously, these applications need to be designed to scale, to expand onto improved and/or additional hardware and to do this transparently (or at the least without having to take down the application for maintenance).

Most small-scale web applications can and are built using off-the-shelf solutions. For a large number of web applications just installing WordPress[6] or a similar content management tool on a web server is good enough. Other small-scale web applications are custom built, but are designed simply for running on a lone web server with no hope of the application being able to handle more than a few thousand users without a serious redesign.

To create a large-scale application such as Flickr[5], YouTube[13] or Facebook[14] without buying all the hardware in the world a well thought out and highly customised design must be produced and followed.

2.1.1 Software Architecture

Cal Henderson describes a layered software architecture in his book *Building Scalable Web Sites* [1]. Henderson's preferred metaphor is that the layered software architecture resembles a trifle, where persistent storage is the sponge and the rest is built up from there. This dissertation will refrain from such a metaphor. A visualisation of Henderson's model can be seen reproduced in figure 2.1, with the left side showing the model itself and the right side showing typical technologies used for these layers in large-scale web applications.

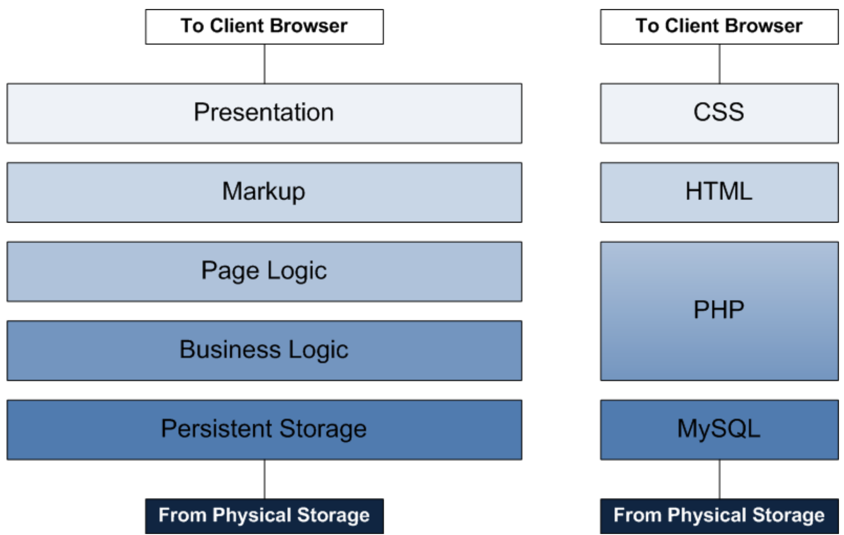


Figure 2.1: Layered Software Architecture for Large-Scale Web Applications

There are two very good reasons for approaching your software architecture from a layered perspective. The first is that separating out the different parts of the code will allow the whole or a part of any layer to be modified to any degree without modifications having to be made to the other layers, as long as the interface to the layers above and below remain the same.

The second reason is that different programmers can work solely on a single layer without having to know about the other layers beyond the interfaces to and from them. A large-scale web application is generally not something that a single developer can create, modify and maintain by themselves. Layering allows developers to devote their work to a single layer and ignore the others (aside from agreeing upon the interfaces to other layers with other developers).

Persistent Storage

Whether its flat files or tables in a database, storage is of the utmost importance. It is where the web application's data is kept and without data to work with the web application is worthless.

There are numerous technologies available for data storage, from database technologies such as MySQL and Oracle to flat file systems. It's important to note that the line between the storage layer and the above layers can become blurred when business and page logic get stored in a database for dynamic execution, such as storing PHP code in a MySQL table to be later retrieved by other PHP code and executed using the `eval` function.

Business Logic

Getting data from where it is kept in storage to the user is an obviously important part of any web application and that's where business logic comes in. Business logic consists of the various algorithms written that manipulate the web application's data and allow the higher layers access to the data. These interactions with the storage layer are what define the web application.

Page Logic

This far up the model there is a web application that has data, which can be accessed and changed in a variety of ways. However, there is no means to describe what data manipulations can happen when. That's where the page logic comes in (Henderson also refers to this as *interaction logic*).

Page logic dictates what data is displayed, the ways in which it can be manipulated and how these manipulations take place.

Keeping page logic separate from the above markup layer can be difficult. One method for doing so is to just keep the code for generating markup completely separated from the page logic, even if they're both implemented in the same language (PHP, Perl, etc.). The onus here to keep the layers separate is completely on the developers, so a strict eye must be kept on the code to ensure the layers don't begin to merge.

An alternate option for separating page logic from markup is to use templating. Setting up the markup layer using a templating system will move the burden of separation of layers from the developers to the code. The page logic will have an interface of what data it can pass up to the template. Using this technique, the markup layer will only be able to work with data explicitly passed up to it from the page logic. This method will reduce somewhat the flexibility of the application, but will make it more modular

and maintainable. There are a number of templating engines available for use including Smarty for PHP [9] and the Template Toolkit for Perl [10].

Markup

Without markup the data presented to the user by the lower levels is unstructured and likely confusing. By adding markup to our web applications (HTML, XML, etc.) we are making the lower layers digestible to the standard user.

Technologies available for the markup layer include HTML and XHTML. Deciding what version of HTML to use has suddenly become a more-than-trivial task. HTML 4.01 has been a standard of development for the last ten years, but HTML 5 compliant browsers are now being developed, such as Internet Explorer 8 and Firefox 3 [7]. However, with around 15% of web users still running the standards-disregarding Internet Explorer 6 as their browser of choice [8], sticking to the widely implemented features of HTML 4.01 may be best for the time being.

Presentation

Despite functionally being less important than the lower layers, the presentation layer of a web application is quite important. A well thought out presentation will mean a user-friendly web application, which will mean more users, which will mean there was a reason for approaching the design from a large-scale viewpoint to begin with.

When building a web application the obvious choice of technology for the presentation layer is CSS. Presentation can be achieved within HTML as well, with tags such as `<center>` and `<font>`, but these tags have become deprecated in standards and this will lead to a very blurred line between the presentation and markup layers.

The Interfaces

As was already discussed, each layer needs to communicate with the layers above and below it. This is what interfaces are for. Both data and control can transfer up and down throughout the layered model through these interfaces. The layers use the interfaces to pass requests and receive responses from one another. For instance, between the storage and business logic layers the interfaces need to deal with reading and writing data.

The most obvious interface in the layered model is between the markup and presentation layers. The markup, written in HTML or XHTML, will have tags with attributes `id` and `class`. The presentation, written in CSS, will use the values of these attributes as selectors for applying style rules (markup tags themselves are also used as selectors). The interface here is the values of `id` and `class`. Developers working on the markup can

modify their code as much as they like without breaking the presentation, as long as they leave the interface untouched. Likewise, developers working on the presentation can also meddle to their hearts' content with their code if they don't modify the interface.

The way these layers can become merged is again obvious and a very easy trap to slip into. Developers working on the markup can begin to eat into the presentation layer if they start to use presentational markup tags such as `<b>` or `<strike>` or presentational tag attributes such as `background` and `color`. CSS can also be written directly into the markup using the `style` attribute, which is yet another way of destroying the separation between the two layers.

The interface between business logic and page logic can be difficult to define in some cases. If the business logic is written in C++ and the page logic in PHP, then the separation of the layers is already there. However, if both layers were written in PHP it once again becomes all too easy to blur the line. A set of functions needs to be defined to create an interface between the two separated layers of PHP and various schemes will need to be created to provide agreed upon names for functions and data objects.

2.1.2 Hardware

Of course, the software is not the only part of a large-scale web application. The hardware that a web application runs on is a monumentally important part of it when dealing with large-scale applications. Web applications such as Gmail, YouTube and Flickr run across thousands of machines, with different machines for different tasks (web servers, database servers, etc.).

Google was estimated to be running their various web applications on 450,000 machines built using commodity components in 2006 [11]. Using off-the-shelf commodity components and machines can be a good idea, especially at the early stages in the life of a web application. The hardware is much more flexible and affordable. As a web application grows on its initial commodity hardware, it can be deduced what hardware needs to be improved such as learning that nodes need more memory or faster processors.

However Google seem to have taken on the idea of buying commodity hardware as their creed. In 2003 their search application was running on 15,000 machines assembled from commodity components, all equivalent to mid-range desktop PCs with a larger amount of storage space [12]. This set up is far more cost-effective than the traditional method of buying high-end servers (in a smaller number).

Platform Options

When devising what platform to run your web application on there are several options you can take, depending on the scale. You could just run your application on your local machine, but this isn't really going to handle many users, no matter how well designed your software is. The next step up from this is to rent space from a shared hosting service. The web application would then be sitting on a machine in a data centre (DC) that is shared with other customers. It is difficult in these situations to sustain consistent performance, as the amount of resources available to your application can vary wildly based on the actions of the other users on the server.

A step up from this is running the web application on a dedicated server. With dedicated hosting you have total control over what is running on the machine. New software and modules can be installed on the machine and the OS can be tweaked with to improve performance. Of course, the machine is still sitting in a data centre somewhere owned by the company running the DC. Upgrading the hardware of the machine may be difficult if not impossible.

Once a web application reaches a size where complete control over all aspects of the hardware is needed, it's time to move to a co-location facility. These facilities (colos) are run by companies who supply the space and power, while their customers provide the hardware and their own maintenance. This is a great set-up for applications that require dozens or even hundreds of servers cost-wise. However, migrating from one colo to another can be a gargantuan task that may not even be cost-effective.

The final and largest option is to set up a dedicated data centre when a web application has grown to such a scale as to require thousands of servers. Obviously running a data centre is an expensive task, hardware aside. Personnel, a facility designed to house a data centre and bandwidth are but a few of the initial and on-going costs of operating a dedicated DC.

These aren't all of the options for where to run your web application. Recently cloud computing services have emerged as a viable alternative to the above options. Running a web application on a cloud computing platform, such as IBM's BlueCloud, Google's AppEngine or Amazon's Elastic Compute Cloud, offers a great potential for an application to scale to millions of users without the up-front hardware costs. Of course, any web application could potentially reach the point where commercial cloud services will become more expensive to use than even a dedicated DC. Cloud computing services will be discussed later in this dissertation.

2.1.3 Scaling

Scalability as a term can be difficult to define, especially given the many incorrect and obtuse definitions that are used. The Linux Information Project has provided such an obtuse definition [2]. It defines a scalable system to be one in which, "the throughput changes roughly in proportion to the change in the number of units of or size of the inputs."

A better definition is provided by Cal Henderson in his book *Building Scalable Web Sites*. He defines a scalable system to be one that can accommodate usage increases, can accommodate data increases and is maintainable [1].

Approaches to implementing scalable applications can be broken down into two categories: Vertical scaling, or scaling-up, is the process of deploying an application on a number of large, powerful machines; Horizontal scaling, or scaling-out, is the process of deploying an application on a larger number of smaller machines.

Traditionally these types of applications were deployed using a vertical scaling solution. Hardware companies such as HP, IBM and Sun focused their efforts on creating more powerful commercial servers with higher and higher clock rates [3]. In the last ten years or so web-based companies such as Google and Amazon began to use horizontal scaling solutions. These clusters were seen as the only workable approach to having the computational power needed by these giants.

One compromise between the two approaches is "scale-out-in-a-box". In this case a vertical scaling solution machine is configured to have multiple instances of the application run on it concurrently. This approach has seen an improvement over a purely vertical scaling approach, as shown in Michael et al.'s experiment with a search application [3].

In their experiment, Michael et al. show that horizontal scaling solutions have a performance advantage over vertical scaling solutions when dealing with search applications. Search applications, theirs using the Nutch/Lucene framework, are highly parallelisable and thus suited to the cluster based approach.

A disadvantage of horizontal scaling approaches is that the larger number of machines increases the complexity of network management. Additionally many applications are not very parallelisable and thus not suited to being distributed. In a purely horizontal scaling approach a point may be reached where adding new hardware will not increase performance, perhaps even decreasing it. When an application reaches a performance plateau with horizontal scaling it is best to begin to scale vertically by replacing existing machines with more powerful ones.

2.2 Cloud Computing

Cloud Computing has come into being in the last few years and is the new buzzword of the technology industry. Of course like other technological buzzwords such as Web 2.0 everyone seems to have their own definition of what cloud computing actually is. To some cloud computing is the next step in utility computing while others see it as the direction that web applications are going.

Weiss describes a number of different "cloud shapes" in his paper [15]. He describes the data centre, distributed computing, utility grids and software-as-a-service (SaaS) as different models of cloud computing. Vaquero et al. attempt to pin down a suitable definition for clouds that describes how they differ from grids. Their proposed definition is thorough, but verbose:

> "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs." [47]

In creating their definition, Vaquero et al. studied definitions from numerous experts, which featured many attributes of cloud computing such as immediate scalability, optimal usage of resources, pay-as-you-go pricing models, and virtualised hardware and software.

2.2.1 Everything as a Service

Lenk et al. describe a stack architecture model for the various types of cloud computing services [48]. This stack model is recreated in figure 2.2, which also shows examples of services offered at each level of the stack.

Infrastructure as a Service (IaaS)

IaaS is the level of the stack closest to the hardware and comes in two varieties, physical resource set (PRS) services and virtual resource set (VRS) services. PRS services are specific to the underlying hardware where as VRS services can use virtualisation to provide a homogenous infrastructure for developers. Examples of VRS services include Amazon Web Services (AWS) such as Amazon EC2 and Amazon S3 [16] and EUCALYPTUS[1], an open-source implementation of AWS [49].

[1] Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems

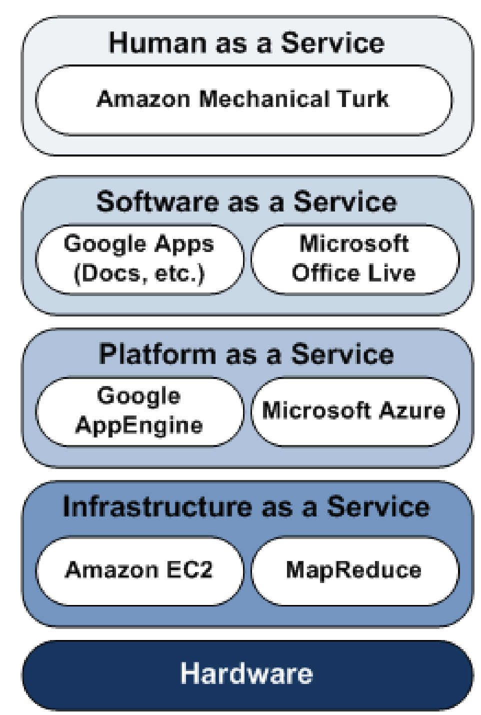


Figure 2.2: Cloud Computing Services Stack

Platform as a Service (PaaS)

PaaS is the next level up the stack and can be broken down into Programming Environments and Execution Environments (which generally contain a Programming Environment). An example of a Programming Environment is the Django Python framework [50]. Django is used in the Google AppEngine [31], which is an Execution Environment. Microsoft's Azure Services Platform, an Execution Environment, allows developers to use a wide-variety of programming languages as the Programming Environment.

PaaS services need not be supplied directly from the company that owns the cloud infrastructure. Any enterprising developer using an IaaS service, such as Amazon Web Services, could develop their own PaaS service on top of it.

Software as a Service (SaaS)

For the most part when an end-user is directly using a cloud-based service, it's SaaS. Examples of SaaS include the Google Apps range (Gmail, Google Docs, Google Calendar, etc.) [51], Microsoft's Office Live [52] and Salesforce.com [53]. SaaS can also be what Lenk et al refer to as Basic Application Services, such as the Google Maps API. These services can be used to create mash-up web applications such as WikipediaVision [54], which shows the location of people editing Wikipedia articles around the world in near real-time.

Human as a Service (HaaS)

The top-most layer of the cloud stack is HaaS, which is subdivided into two categories: Crowdsourcing (CS) and Information Aggregation Services (IAS). Amazon's Mechanical Turk [55] is an example of a CS service, where developers are able to deploy jobs that a computer cannot perform (at all or easily) to people who sign up to the service to earn money completing these tasks. An example of an IAS service is Digg [56], where human users post interesting news stories, images and videos and "digg" the content they enjoy.

2.2.2 Commercial Cloud Computing Services

There are a number of companies offering cloud computing services to customers. These companies offer platform as a service (PaaS) and infrastructure as a service (IaaS), which allows developers and start-up businesses to create applications and services for their own customers without the massive overhead of purchasing hardware to run it on. It's no surprise that PaaS and IaaS offerings are coming from such giants as Google [28], Microsoft [29] and Amazon [16]. These companies already have the necessary hardware infrastructure for hosting these cloud-based services.

One fear of using these commercial services is vendor lock. This is the situation in which a developer writes their application or service for a particular cloud service, but finds that their code won't work on other cloud computing services and are then stuck using that particular service [34]. These fears have somewhat been alleviated by frameworks such as AppDrop [35] and AppScale [36], which allow applications written for the Google AppEngine to be run on IaaS systems such as Amazon Web Services.

Google AppEngine

Google announced a preview release of Google AppEngine (GAE) in April 2008 [30] telling the world that developers could now create a web application on Google's infrastructure with the same ease that one can create a blog using their Blogger service. One of Google's selling points for it is that if an application is using resources under a certain quota there is no cost to the developer [37].

The PaaS service offered by GAE is somewhat more limited than other cloud-based services available. GAE only supports applications written in Python or Java (or Java Virtual Machine based languages such as JavaScript and Ruby). However, the trade-off for this lack of flexibility is a lack of complexity. Google handles load balancing for GAE applications and automatically scales them, meaning that the developer doesn't have to worry about ensuring enough resources are available to their application when needed

[31].

GAE uses a non-relational database for storing application data called the datastore. The datastore is schemaless, meaning that if two objects are of the same type they are not obliged to have the same properties or value types. Queries made to the datastore are written in an SQL-like syntax called "GQL". [38]

Applications on GAE run within a secure sandbox. Applications can only be communicated with over the Internet via HTTP and HTTPS and applications must use the provided URL and email services to communicate with other machines. Applications also cannot write to any storage space on the machines they are executing on; they must use the provided storage services such as the datastore and memcache. [31]

The AppEngine also features integration with a number of other existing Google services. Applications on GAE can use Google Accounts for users to sign into their apps, instead of having to build their own user authentication and management system [32]. Applications can also send emails using Google's existing mail infrastructure [33].

Microsoft Azure Services Platform

The Azure Services Platform is Microsoft's PaaS cloud services platform, which offers developers a cloud-based operating system known as Windows Azure and various other cloud-based services such as SQL Azure (a cloud-based relational database service based on Microsoft's SQL Server) and Live Services (a service for managing user data that's tied into Microsoft's Windows Live) [39].

Unlike Google AppEngine, Azure offers developers the opportunity to use a broad range of programming languages, from PHP to C#. Storage on Windows Azure is separate to the SQL Azure service. Windows Azure storage is non-relational, instead it allows application to store binary large objects (BLOBs), queues for communicating components and a simple non-relational database similar to Google AppEngine's datastore and Amazon's SimpleDB [40].

Azure offers more control to the developers than Google AppEngine. Applications on Azure have a configuration file associated with them to control their behaviour. Using these configuration files the developer can set the number of instances that the underlying Azure OS should run on.

One of the more attractive offerings from Azure is its Live Services capability. It allows applications to access data stored on various Windows Live services, such as Windows Live Messenger, Hotmail and Bing. This allows developers to build web applications that integrate with existing Windows Live applications and thus pull users from an existing pool [40].

2.3 Amazon Web Services

Amazon Web Services (AWS) is a group of cloud-based services provided by Amazon, which they advertise as being cost-effective and flexible. AWS has services such as Amazon Elastic Compute Cloud (EC2), for processing, and Amazon Simple Storage Service (S3), for data storage. AWS differs from traditional hosting in that instead of users renting or buying equipment and having resources go under-utilised Amazon only charges the user for the resources that they use [16].

Charges for the various AWS services are based on different metrics for each service. For instance, for EC2 charges are applied for every hour an EC2 server instance is running and for data transferred in and out of AWS, while for S3 charges are applied for data storage on a monthly basis and for data transfer and operations on the data. The charges for AWS are fine-grained so that users are paying for exactly what they are using. Using S3 for storage, for example, costs $0.18 per GB per month in Europe and the price per GB drops with more data stored ($0.15 per GB per month for all data after the first 500 TB), so the pricing model is very good for large-scale web applications among other uses.

AWS services are all designed with scalability for the user as a high priority. If a user's application running on EC2 suddenly needs more processing resources (i.e. more EC2 server instances running), this can be achieved in minutes, not hours or days, and it can be achieved automatically using other AWS services such as Auto Scaling and Elastic Load Balancing.

2.3.1 Amazon Elastic Compute Cloud (EC2)

Amazon EC2 is a cloud-based processing service on AWS. Users of EC2 can launch and terminate server instances on a scale of minutes, rather than the hours, days or weeks of traditional hardware solutions. The main concept behind EC2 is that of server instances [17].

There are a number of different types of instances that users can choose from and they are divided into two categories: standard and high-performance. Each type has several subtypes, with various levels of processing power, memory and on-board storage.

Because AWS is built on top of heterogeneous hardware processing power is defined in Amazon EC2 Compute Units as a standard. One EC2 Compute Unit has the equivalent CPU capacity of a 1.0 - 1.2 GHz 2007 Opteron or 2007 Xeon processor [18].

The storage provided on an instance (referred to by Amazon as "instance storage") is volatile. Data will survive the instance rebooting, either intentionally or accidentally, but it will not survive the underlying hard drive failing or an instance being terminated. Amazon encourages developers to use their Simple Storage Service (S3) as a persistent

storage solution.

EC2 Instances are created by launching machine images known as Amazon Machine Images (AMI). AMIs contain the operating system that will be launched on the instance along with all the software for that instance and its configuration. AMIs are stored on Amazon S3 and Amazon provides a number of pre-bundled public AMIs (with Linux, UNIX or Windows as the OS) that can be launched by users without any configuration.

Users can also create their own custom AMIs (private AMIs), either from scratch or using a public AMI as a base. Private AMIs are created by a process called bundling, in which a machine image is compressed, encrypted and split, the parts of which are then uploaded to Amazon S3.

Every EC2 instance has two addresses: a private IP address (resolved to by a private DNS name) that can only be reached within the AWS network; and a public IP address (resolved to by a public DNS name) that is reachable from the Internet. The addresses are mapped to each other through Network Address Translation.

Instances can also be assigned Elastic IP addresses, which are static IP addresses designed for cloud-based services. Elastic IP addresses are assigned to the user's AWS account and can then be mapped to instances dynamically. Every elastic IP can be associated with one EC2 instance and can be quickly changed over to another instance in the event of a failure.

The EC2 network is comprised of Regions and Availability Zones. Regions are groups of EC2 machines in different physical locations e.g. the US and Europe. Availability Zones are groups within regions that are designed to be isolated from problems and failures in other Availability Zones, but can still easily communicate with them. Regions however are completely independent from each other.

EC2 provides mechanisms for ensuring proper network security even with instances booting and terminating at any time. Security Groups are defined by the user to be a set of access rules. Instances can then be added to and removed from the security groups. Any particular instance can be a member of any number of security groups at any given time. By default EC2 instances are assigned to the security group `default` in which all network traffic from outside the `default` group is discarded.

Amazon EC2 provides developers with two APIs for interacting with the service. One is the Query API in which operations send data using GET or POST methods over HTTP or HTTPS. The other is the SOAP API in which operations send data using SOAP 1.1 over HTTPS.

Amazon Elastic Load Balancing

Amazon Elastic Load Balancing is a service for use with Amazon EC2 to provide additional scalability and availability. The main concept behind Elastic Load Balancing is the LoadBalancer. This is represented by a DNS name and a set of port numbers. This DNS name can then be mapped to a user-specified domain name and used as the point-of-entry to whatever the user is running on EC2 [21].

The LoadBalancer has a number of EC2 instances assigned to it by the user (all of which must be in the same EC2 region) and distributes all load directed to it between these instances. An overview of the architecture of Elastic Load Balancing can be seen in figure 2.3.

The LoadBalancer also monitors its registered EC2 instances and will stop directing traffic to any instances that becomes "unhealthy".

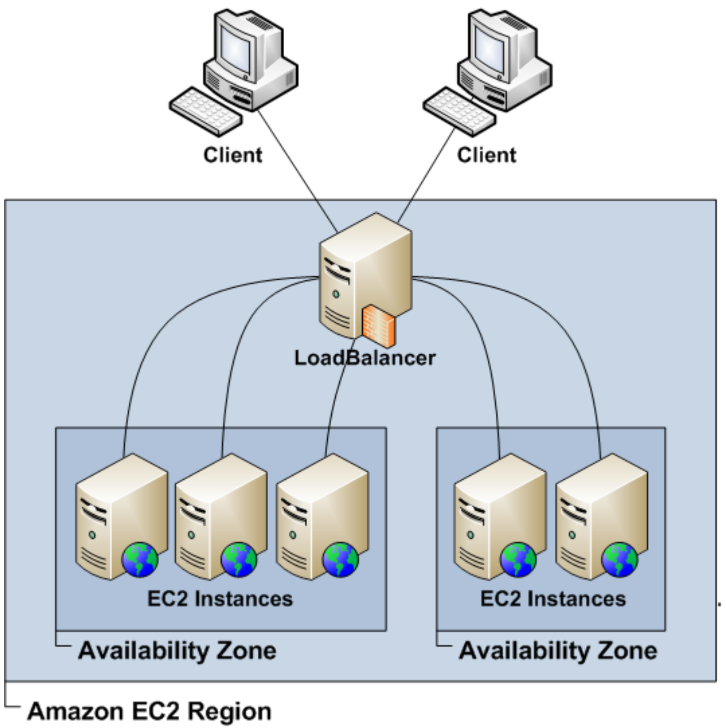


Figure 2.3: Elastic Load Balancing Conceptual Architecture

Amazon CloudWatch

Amazon CloudWatch is a service for use with EC2 and Elastic Load Balancing. CloudWatch gives AWS users the ability to monitor the performance of their EC2 instances and LoadBalancers by providing them with metrics of various properties such as CPU utilisation and disk usage. These metrics are aggregations of data retrieved over one minute

periods [19].

Metrics have various properties such as their value and units as well as statistics such as maximum and average. Every metric also has a dimension, which specifies what the metric has been aggregated over such as a particular EC2 instance, an Availability Zone or an Auto Scaling Group.

Amazon Auto Scaling

Auto Scaling is another service for EC2 that can automatically scale up or down the amount of EC2 instances running based on user-specified parameters, such as traffic statistics or memory utilization. Auto Scaling monitors its assigned EC2 instances and will launch replacement instances in the event of a crash [20].

Auto Scaling works by having the user create an Auto Scaling Group consisting of multiple EC2 instances within a single Availability Zone. The user attaches a Launch Configuration to this group, which specifies parameters for new EC2 instances in the group. The user then defines Triggers based off metrics from Amazon CloudWatch. These triggers control when the Auto Scaling Group should scale up or down.

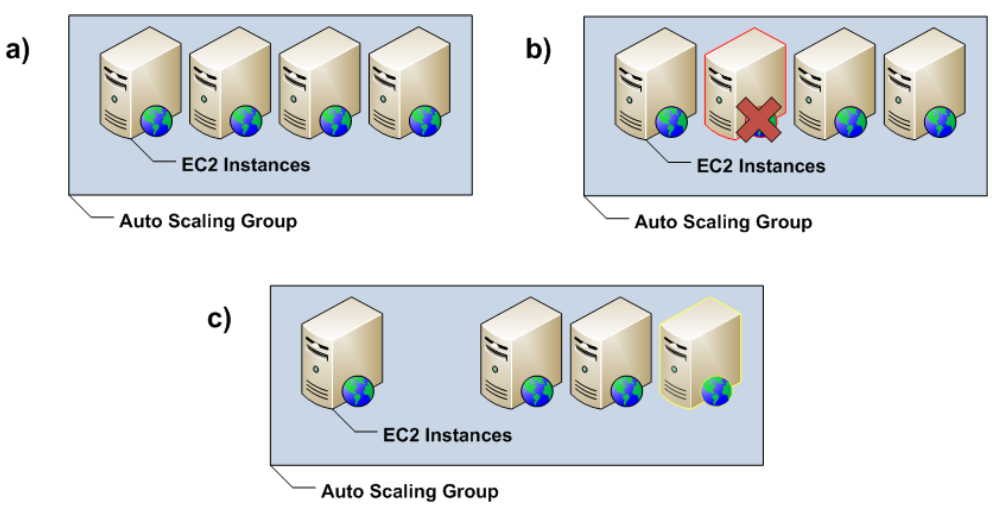


Figure 2.4: Auto Scaling Example

Figure 2.4 shows an example of Auto Scaling in action. The Auto Scaling Group in 2.4.a) is configured to keep four EC2 instances running for the particular load level on the group. In 2.4.b) one of the instances crashes unexpectedly. In 2.4.c) Auto Scaling launches a new EC2 instance using the group's Launch Configuration.

2.3.2 Amazon Simple Storage Service (S3)

Amazon S3 is a cloud-based storage service within AWS. It is designed to make storing and retrieving data on AWS as simple as possible. Data is stored using a straightforward flat model, on top of which users can build their own storage structures using hierarchies. S3 also features a simple yet versatile access control system, where objects can be made private, public or have access granted to certain groups of users [22].

The two main concepts that S3 is built from are buckets and objects. Buckets are containers for data objects. All objects stored on S3 are stored in buckets.

Each bucket has a name that is completely unique within S3. Bucket names are directly mapped to URLs for addressing data stored on S3. If a bucket is named `foundry` then it can be addressed with the URL `http://foundry.s3.amazonaws.com`. This URL can be appended with the name of an object to create an address for any object stored on S3. There are a number of restrictions on bucket names and additional guidelines that should be followed to conform to DNS requirements.

Every bucket is owned by only one AWS user and buckets cannot be transferred to other AWS accounts, however access to objects in a bucket can be granted to other users and to agents outside AWS. Every AWS user can have a maximum of one hundred buckets at any given time. Buckets cannot be nested, only objects can be stored inside buckets.

The other main concept of S3 is objects. An object consists of four components: a value (the data being stored in that object), a key (the unique identifier for that object), metadata (additional data associated with the object) and an access control policy.

The object value, being the data a user is storing in that object, can be between one byte and five gigabytes. There is no limit to the number of objects that a user can store on S3 and no limit to the number of objects that can be stored in a bucket.

The key is the name of the object and must be absolutely unique within the bucket that contains the object. Keys can be any size from one byte to 1,024 bytes. Keys can be listed by their bucket and a prefix. This allows users to use common prefixes to group together their objects into a hierarchy, meaning that the flat storage model of S3 buckets can then be turned into a directory-like model for storing data. Object keys can also be given suffixes, like `.jpeg` or `.mpeg`, to make the key more descriptive of what the file is, though this is not a requirement of S3.

Say an AWS user is using S3 to store data related to different pre-built computers. They may name their objects like in the following scheme:

```
Desktop/Dell/Dimension/2350
Desktop/Dell/Optiplex/GX270
Laptop/Dell/XPS/M1530
```

`Desktop/Apple/iMac/A1225`

`Desktop/Fujitsu Siemens/Scenic/P300`

Listing the objects in this bucket by using prefixes will allow the user to interact with the data in a directory-like manner. If the user lists using the prefix '`Desktop/Dell/Dimension`' they will be returned a list of objects with that prefix, which corresponds to a list of models of PC manufactured by Dell under the Dimension brand.

If the user lists using the prefix '`Desktop/Apple`' they will be returned a list of objects that correspond to the different desktop models produced by Apple, such as iMac and Mac mini, but it will nest together all objects below that in the hierarchy. This means that only the product lines' names are returned and not the model numbers below them.

The metadata of an object is a set of key/value pairs and is divided into two types: system metadata and user metadata. System metadata is used by S3 while user metadata can be any key/value pair defined by the user. User metadata keys and values can be any length, as long as the total size of all metadata (system and user) for an object is less than two kilobytes. Encoding of values for metadata depends on which API is being used. UTF-8 must be used with the SOAP API, while US-ASCII must be used with the REST API.

Access control on objects is managed by access control lists (ACL). Every object, as well as every bucket, has an ACL. When a request is made to S3 it checks the ACL of the object or bucket to check if the requester has been granted permission. If the requester is not authorised to access the object then an error is returned by S3. There are a number of different types of groups that can be granted permissions and a number of different permissions, such as `READ`, `WRITE` and `FULL_CONTROL`.

S3 provides two APIs for making requests, one using REST and the other SOAP. The REST API uses standard HTTP headers and status codes, with some additional headers added in by S3 to increase the functionality. The SOAP API uses SOAP 1.1 over HTTPS and Amazon S3's WSDL file located at [23].

Updates to data in S3 are atomic (i.e. the entire operation takes place or it rolls back to its state before the operation began). Data on S3 is replicated across multiple servers to ensure availability and fault-tolerance, but this does mean that updates need to propagate across the various replicas. This can mean that a read performed soon after a write may return the old value of the object instead of the new one.

2.3.3 Amazon SimpleDB

Amazon SimpleDB is a cloud-based databasing service within AWS. SimpleDB removes the complexity of large-scale databasing by providing a simple-to-use API to the user that

is scalable and fault-tolerant. It also removes the up-front costs of purchasing a relational database cluster and on-going costs of maintenance and administration by using the AWS model of paying based on utilization [24].

SimpleDB's data model is, abstractly, quite like a standard relational database. The database in an RDB is equivalent to the AWS user account in SimpleDB. The tables of an RDB are equivalent to domains in SimpleDB. The columns of a table in an RDB are equivalent to the attributes of a domain in SimpleDB and the rows of an RDB table are equivalent to the items of a domain in SimpleDB.

There are a few differences between SimpleDB and standard RDBs. Firstly, an item/attribute intersection in SimpleDB can have multiple values. As well, an item in SimpleDB does not need to have a value for every attribute (an RDB, such as MySQL, would store a `NULL` value in place of a row/column intersection with no value). This means that sets of data with completely different attributes can be stored in the same SimpleDB domain.

Similarly to Amazon S3, SimpleDB replicates user data transparently across multiple locations. This leads to consistency issues as with S3, where a write followed quickly by a read can return an old value from a server that has yet to have the new value propagated to it. Amazon refers to this as "eventual consistency".

There are a number of limitations on SimpleDB. Domains are limited to 10 GB in size, with a maximum of one billion attributes per domain. AWS users are also limited to 100 domains. Attributes, items and values are all limited to 1,024 bytes and items can have a maximum of 256 attribute-value(s) pairs. There are further limits imposed on the use of `SELECT` statements.

Partitioning data across a number of domains can lead to improved performance and there are a number of different ways for a user to partition their data. Sometimes data will naturally have some logical way to partition it e.g. data about computers could be partitioned into data about desktops, data about laptops, data about mobile devices and so on.

For data that doesn't provide such a natural partitioning, it can be partitioned using a hash function (e.g. MD5) of the item name. The data can then be partitioned based on the last n bits of the resulting hash value into 2n domains.

Like Amazon S3, SimpleDB has both a REST API and a SOAP API for making requests to the service. Both APIs provide SQL-like `SELECT` statements that include restrictions via `WHERE` and `LIMIT` expressions as well as sorting via `ORDER BY` expressions.

All data in SimpleDB is stored as UTF-8 encoded strings. This means that to perform proper comparisons between numerical values and dates they must be represented identically. Numerical values must be zero-padded to the largest number the user will store. If

the user wishes to store negative numbers then all numbers must be offset by the largest negative number the user will store. For dates Amazon recommends using the ISO 8601 format e.g. `2009-09-11T17:00:00TZD` [25].

SimpleDB also provides the user with utilisation data called BoxUsage values that can be used to optimise database performance.

2.3.4 Amazon Simple Queue Service (SQS)

Amazon SQS is a cloud-based distributed queue system within AWS. In a system using SQS (or distributed queues in general) a number of the components are producers, while another group of the components are consumers. Producers add messages to a queue in the system and consumers remove the messages from the queue. The queue(s) act as a buffer for messages between components, which is helpful if producers output messages faster than consumers can process them [26].

SQS provides queues that can be read from and written to by multiple consumers and producers. Every message on a queue in SQS is transparently replicated across multiple servers. (Note: each message is replicated independently from other messages on the queue, so each server has only a subset of the entire queue.)

SQS guarantees that every message will be delivered at least once, but it does not guarantee first-in, first-out ordering of messages. It also does not guarantee a message delivery for every request if the queue has a small number of messages on it (less than one thousand).

Unlike other AWS services there are no limits on the number of queues a user can have or the number of messages in them. However, messages will only remain in a queue for four days before being deleted and inactive queues will be deleted after thirty days.

SQS queues have several identifiers associated with them, each with a different use. Queue URLs are based off a name given to the queue by the user that must be unique with the AWS user's own scope. Queue URLs take the following form:

```
http://queue.amazonaws.com/AWS-ACCOUNT-NO/QUEUE-NAME
```

The Queue URL is used to identify a queue when performed operations on it. Message IDs are a system-assigned ID returned to consumers with messages that were used for deleting messages from the queue in older versions of SQS, but as of API version 2009-02-01 the ID can no longer be used for this purpose. Instead consumers receive a Receipt Handle with every message that is associated with the act of receiving the message and not the message itself. This is what the user must use to delete messages from the queue.

Since the act of receiving the message does not delete it from the queue, this means that the message remains until explicitly removed by the consumer that received it. To ensure that other consumers do not also receive the message before it is deleted SQS provides a visibility timeout for every message. Once a message is received by a consumer a timer is started. While this timer is running no other consumers can retrieve that message from the queue, but the receiving consumer can still see the message and delete the message before the timeout. Once the timeout is reached the message becomes visible to other consumers again. The visibility timeout can also be extended or reduced on the fly.

Like other AWS services SQS provides a SOAP API and a REST API for performing operations on queues.

2.3.5 AWS Security

The nature of AWS means that approaches to security are quite different from those of a traditional data centre. Amazon's white paper on the security processes of AWS [41] begins with a description of Amazon's physical security. Some of the physical security measures implemented by Amazon include housing AWS data centres in non-descript facilities, using "state of the art intrusion detection systems" and applying rigorous authentication procedures to authorised personnel entering the data centre floors.

All server accesses by AWS administrators are logged and administrators have absolutely no access to the virtual OSs being run on EC2. Access logs are regularly audited by Amazon. Amazon also provides protection against traditional network security attacks. Distributed Denial of Service (DDoS) attacks are mitigated using the same techniques as Amazon's main website. Man-in-the-Middle attacks are prevented by using SSL to provide server authentication. EC2 instances are also prevented from sending traffic with spoofed IP or MAC addresses. Other AWS customers cannot sniff traffic not addressed to them, thanks to the AWS hypervisor.

Both S3 and SimpleDB provide robust user access control with Access Control Lists. Transfers to and from S3 and SimpleDB can be done using SSL. S3 does not encrypt the data it stores, but users are free to encrypt their data before sending it to S3 for storage. When data is deleted from S3 and SimpleDB the mapping between the public name of the data and the data itself is removed from across AWS in a matter of seconds and the deleted data will then be overwritten by new data.

2.4 User-Generated Content

One of the big trends of *Web 2.0* is User-Generated Content (UGC), text, images, videos and other media created and uploaded by the user base of a website. UGC has advantages over traditionally generated content in that traditional content is created and modified by a small group of people running a website, whereas UGC is created, modified and consumed by the entire user base of a website. Ramakrishnan et al estimated the amounts of content create on the web daily [66]. They found traditional web content increases at a rate of around two gigabytes a day, while user-generated content increases at a rate of eight to ten gigabytes a day.

Of course, with that much content being churned out daily, how can the quality content be separated from the useless content and spam? Usually sites rely on the users themselves to conduct quality control and hope that the collective intelligence of the user base will win out. Wikipedia has thousands of dedicated users that moderate the encyclopaedia themselves. As of August 2009, Wikipedia contains nearly three million pages of user-generated content and that's just the English language site [67]. On those pages there have been over 300 million edits since Wikipedia was founded.

YouTube claim to have hundreds of thousands of videos uploaded daily, with ten hours of video being uploaded every minute [68]. Of course, a lot of content uploaded to YouTube is duplicates of existing content or content that violates copyright law.

Cha et al estimate that most videos on YouTube have one to four duplicates, with some having upwards of one hundred duplicates [69]. Holt et al estimate that nearly ten percent of content uploaded to YouTube is removed due to copyright violations [70], but Cha et al report this figure to be much lower with only 0.4% of content being removed and only five percent of that being removed due to copyright violations.

2.4.1 Content Processing

UGC such as images and video can be processed into "new" content by applying transforms to the content. Tools for doing this on a client already exist with applications such as Adobe Photoshop [71] and GIMP [72] for transforming images and Adobe Premiere Pro [73] and Apple's Final Cut Pro [74] for transforming videos. Of course, these tools are not much help when trying to modify UGC programmatically or in bulk.

Some web applications provide users with the ability to perform certain transforms on their UGC, by abstracting certain operations of content processing applications. Facebook provides a few image transforming functions to the user, such as the ability to crop a thumbnail version of their profile picture for use in news feeds and the ability to rotate photos uploaded to albums to fix their orientation. YouTube provides video owners with

the ability to swap out the audio track of their video and replace it with a new piece of audio and to overlay annotations, subtitles and captions on top of their videos.

2.4.2 ImageMagick

ImageMagick is an open-source application for editing and converting raster images. ImageMagick does not have a graphical user interface, but instead is designed to be run from the command line or via various APIs [57]. It can read and write images in over one hundred different formats, including `JPEG`, `PNG` and even `PDF` [58].

There are many different interfaces to choose from to use ImageMagick with various programming languages, including APIs for C, C++, Java, .NET, PHP, Perl and Ruby [59]. There are three existing APIs for using ImageMagick with PHP: MagickWand for PHP [60], IMagick [61] and phMagick [62].

ImageMagick features a large number of different transforms that can be applied to images using the `convert` and `mogrify` functions. Images can be resized, cropped, flipped and rotated [63]. Images can also have a number of different filters applied to them, such as applying a coloured tint to the image, applying a blur to the image or applying an edge-detection filter to the image.

Another interesting transform in ImageMagick's available functions is the -liquid-rescale function, which uses seam-carving to scale images without distorting them. Seam-carving, also known as Content-Aware Image Resizing, can expand or reduce an image by inserting or carving "seams", which are connected paths of low energy pixels in the image. By using seam-carving an image's aspect ratio can be changed without the objects in the image becoming distorted. Seam-carving can also be used to remove certain objects from an image entirely [64].

ImageMagick is not limited simply to editing images; it has support for various video formats such as `AVI` and `MPEG` [58]. With these formats ImageMagick can transform a short `MPEG` video clip into an animated `GIF` or take an interlaced frame from a video file and de-interlace it [65].

2.5 XML and Schema Languages

Extensible Markup Language (XML) is a text format based on SGML and was designed to be a human-readable format for storing data [42]. It has taken off as a standard for exchanging data between applications. All XML documents must be well-formed, that is they must completely comply with a list of syntactic restraints as listed in the XML specification [43]. Languages based on XML include XHTML, SOAP, RSS. Many word

processing tools now store their documents as XML, such as Microsoft Office (Office Open XML) and OpenOffice (OpenDocument).

XML documents consist of elements, which are composed of tags and content (which can contain other elements). XML tags are similar to tags in HTML, in that there are opening and closing tags as well as self-closing tags all of which can contain attributes (name/value pairs). An example of a typical XML element is shown below:

```
<Computer type="desktop">
   <Manufacturer>Dell</Manufacturer>
   <Model name="Dimension" number="2350" />
</Computer>
```

This element has one attribute and two child elements, one of which is composed of a self-closing or empty-element tag with two attributes. As can be seen, XML is quite human-readable.

XML schemas are used to define types of XML document. A schema imposes a set of rules on the structure and content of an XML document. If an XML document conforms to these rules it is said to be a valid document for that schema. Document validation is separate to well-formedness, which defines if a document is or isn't XML.

An XML schema language is a language used to define a schema for XML documents. A number of schema languages exist such as XML Schema (XSD), Document Type Definition (DTD) and RELAX NG. A comparison of the various schema languages found languages such as XML Schema and Document Structure Description (DSD) to have a lot of expressive power, but less expressive schema languages such as DTD are much easier to learn and use [44].

2.5.1 XML Schema

XML Schema is an XML schema language, which is more commonly known as XSD (XML Schema Document) due to the use of "`xsd`" as the XML namespace and "`.xsd`" as the file extension of XML documents written using XML Schema. XSD is based on DTD and a number of other early schema languages. XSD provides more expressiveness than DTD, including the ability to recognise XML namespaces and the ability to specify a type for data in XML elements [45].

XSD provides nineteen different primitive data types (including various boolean, string, number, URI, time and date types) as well as allowing custom data types to be constructed from these primitives. These types can be created by either restricting the values of an existing type, listing an allowed set of values or combining the sets of values from multiple existing types. XSD also provides a number of these derived types in its specification [46].

An example of an XSD definition is shown below:

```
<xs:schema elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="Computer">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Type" type="xs:string" />
            <xs:element name="Manufacturer" type="xs:string" />
            <xs:element name="RAM" type="xs:decimal" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

The following is an example of a piece of XML that would validate under this schema:

```
<Computer>
   <Type>Desktop</Type>
   <Manufacturer>Apple</Manufacturer>
   <RAM>2048</RAM>
</Computer>
```

Chapter 3

Design

This chapter will describe the design of Foundry, the web application for this project. It will describe the requirements, both functional and non-functional, the use cases for the application's end user and the various architectures of the application.

3.1 Requirements

The requirements for this application have been divided into functional and non-functional requirements.

3.1.1 Functional Requirements

1. The application must allow users to upload content to be transformed via a web browser.
2. The application must allow users to specify which available transforms will be applied to their content via a configuration file.
3. The application must allow users to retrieve content, original and transformed, via a web browser.
4. The application must allow users to add and edit metadata to their content via a web browser.
5. The application must allow users to delete their existing content, original and transformed, via a web browser.
6. The application must allow users to add and edit metadata to their configuration files via a web browser.
7. The application must allow users to delete their configuration files via a web browser.
8. The application must allow administrators to add new types of transforms to the application without altering the application's source code.

3.1.2 Non-Functional Requirements

1. The application must be functionally scalable. That is it must be designed in a way that additional content transforms and content types can be added to the application without needing to alter the way existing transforms and content types are dealt with.
2. The application must be load scalable. That is it must be designed to handle increases in the order of magnitude of application users and application data without needing to alter the underlying application code.
3. The application must be fault-tolerant.
4. The application must be reliable.
5. The application must store users' account details securely.
6. The application must be built on a cloud-computing platform.

3.2 Use Cases

Figure 3.1 shows the use cases for this application. Each of the actions carried out by the actor can be broken down into smaller actions carried out within the application. A summary of each use case is also included.

3.2.1 Account-Related Use Cases

Register with Foundry

When the user first navigates to the web application with their browser they have to create a user account. The user is prompted to enter a username and password, along with other registration details. Once the user has completed the registration process they are then able to use the rest of the web application.

Login to Foundry

Once a user has registered they are able to login to the web application via their browser. The application prompts the user for their username and password and if the user enters the correct information they are logged into the application. If the user enters incorrect information the application will not log them in, informs them of their error and allows the user to attempt to login again.

Lookup User Data

Once a user has logged in the application is able to retrieve all data related to that user from storage.

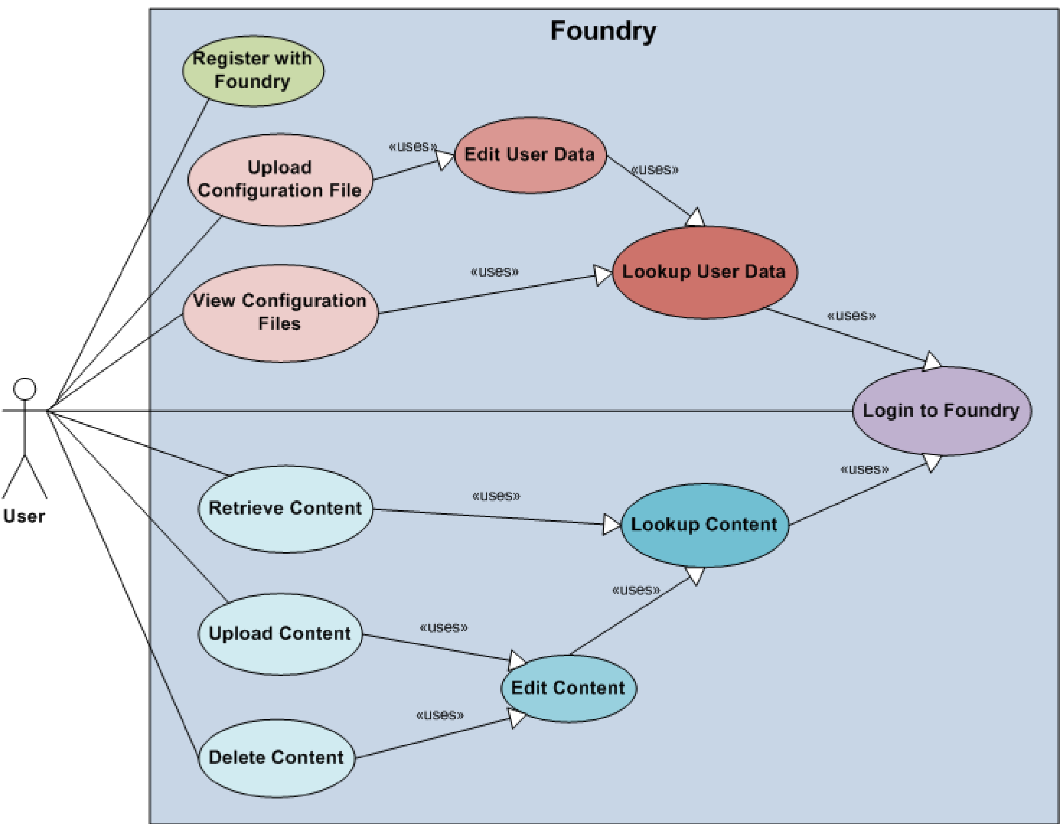


Figure 3.1: Use Case Diagram

View Configuration Files

The user is able to view all of their configuration files for their content transforms that are within the application via a web browser.

Edit User Data

Once a user has logged in and the application has looked up their data it is able to edit the user's data.

Upload Configuration File

The user is able to upload a new configuration file for content transforms to the application via a web browser. If the upload is successful the user is served a web page informing them

that the upload completed successfully. If the upload fails for any reason the application attempts to inform the user of the failure and discards the uploaded file or partially uploaded file.

3.2.2 Content-Related Use Cases

Lookup Content

Once a user has logged in the application is able to look up any content uploaded by and transformed for the user.

Retrieve Content

The user is able to retrieve any original content uploaded to the application and any transformed content processed by the application. This content is served to the user through a web browser.

Edit Content

Once a user has logged in and the application has looked up content related to that user, the application is able to edit and remove that content.

Upload Content

The user is able to upload content to be transformed via their web browser. Before beginning the upload process the user specifies which of their configuration files to use with the content being uploaded. If the upload completes successfully the application serves the user a page informing them of the success. If the upload fails the application attempts to inform the user of this failure and discards the likely partially uploaded content.

Delete Content

The user is able to delete any original content uploaded to the application and any transformed content processed by the application. If the deletion process is successful the application informs the user. If the deletion fails the application attempts to inform the user and attempts to prevent the content from being deleted.

3.3 Architecture

This section describes the functional and technical architecture of the application as a whole as well as a more detailed description of the architecture of the various components.

3.3.1 Functional Architecture

The architecture of the application is shown in figure 3.2. The top layer is composed of the various components of the application, while the bottom layer shows the networks that the components sit on and communicate with each other through. The diagram also shows the various messages that each component sends to and receives from other components.

3.3.2 Functional Architecture Messages

A number of different types of message are passed between components in the application. Each of the messages and their purpose is described below.

Page Request

Page requests are made by the client and sent to one of the web servers via the load balancer. This is the main method for the end user to communicate with the application.

Page Response

Page responses are generated by the web servers and sent to the client. These responses are what allow the application to inform the end user of what it is currently doing.

Content

Content is media provided by the end user (images, videos, text, etc.) to be uploaded to the application for transforming.

Config. File

Configuration files are again provided by the end user to be uploaded to the application. The files tell the application how to handle and transform content uploaded by the user.

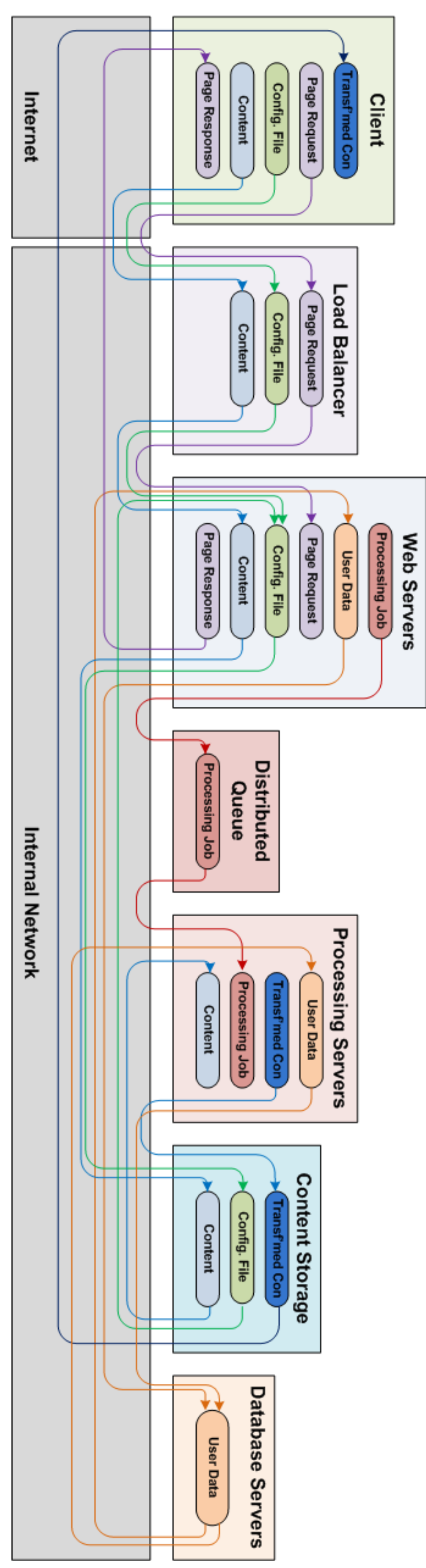


Figure 3.2: Functional Architecture

User Data

User data is determined by the application from information provided by the end user in various page requests, as well as content and configuration file uploads.

Processing Job

Processing jobs are messages passed from the web servers to the processing servers via distributed queues, which inform the processing servers of content that needs to be transformed. Processing jobs contain a reference to the content to be transformed and an instruction of what transform to apply to the content.

Transf'med Con

Transformed content is the result of the processing servers transforming content uploaded by the end user. The content is stored and can be retrieved by the end user via the web servers.

3.3.3 Functional Architecture Components

Aside from the client, the components all reside within an infrastructure as a service (IaaS) cloud computing service and communicate with each other via the service's internal network. The application communicates with the client via the Internet.

Client

The client is the machine that the end user is interacting with the application from. The client makes page requests to the application via the Load Balancer. The client uploads content and configuration files in the same way. The client receives page responses from the web servers. The client is also able to receive transformed content from storage.

Load Balancer

The load balancer is the main point of communication between the end user and the application. The client makes all requests and uploads to the application through the load balancer. When the load balancer receieves a page request, content or a configuration file from the client it chooses a web server to forward the messages on to. The load balancer is responsible for ensuring that load across all web servers in the application is even and that no one web server is being overworked.

Web Servers

The application contains a number of web servers. Each web server deals with communicating with the client and determining user data from interactions with the client. When the client uploads a configuration file, the web server sends this file to the content storage servers and creates new user data for the database servers linking that configuration file to the user that uploaded it.

When the client uploads content to the web server, the web server sends this content to content storage and creates new user data linking that content to the user uploading it. It then retrieves the configuration file specified by the user for processing the content. It analyses this configuration file and creates a new processing job for each transform to take place on the uploaded content that gets sent to the processing queue.

The reason for having the web servers analyse the configuration file and send multiple processing jobs to the queue is one of allowing for functional scalability. Having the individual transform jobs determined by this component leaves the option open to allow the user to specify transforms by means other than a flat configuration file, such as a web form interface or a Flash interface, both of which would be run on the web server.

Alternatives to this design would involve either a single processing server executing all the transforms for a piece of content, which could take a non-trivial amount of time with a large amount of transforms, or to add in an additional component to analyse the configuration file and distribute the individual processing jobs to the processing servers, which could be made redundant or difficult to work with by a new method for the user to specify transforms.

Distributed Queue

The distributed queue is used to line up jobs for the processing servers. The web servers add processing job messages to the queue and the processing servers take these messages off the queue.

Processing Servers

These servers are what actually carry out the transforms in the application. A processing server will take a message off the distributed queue and use this to start processing content. The processing job message contains a reference to the content to be transformed, stored on the content storage servers, and instructions for what transforms to carry out on the content.

The processing server interprets the instructions of the processing job message to run a transform of the uploaded piece of content. It retrieves the referenced content

from the storage servers, then runs the transform on the content before finally sending the transformed content to the storage servers and updating the user data stored in the database servers.

Content Storage

These servers are responsible for storing the original content uploaded by the user, the transformed content created by the processing servers and the configuration files uploaded by the user.

Database Servers

These servers are responsible for maintaining information related to the end users of the applications. This includes the user's authentication data, references to the user's uploaded and transformed content stored on the storage servers and references to the user's configuration files stored on the storage servers.

3.3.4 Technical Architecture

Figure 3.3 shows the technical architecture of the application.

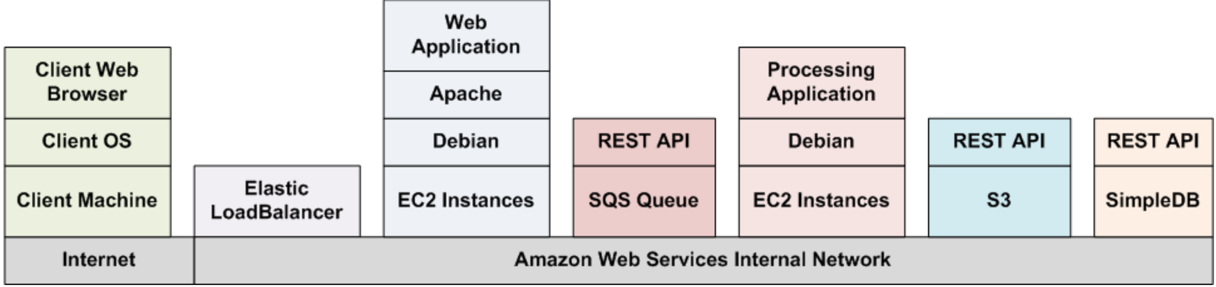


Figure 3.3: Technical Architecture

The IaaS service being used here is Amazon Web Services [16]. The web servers and processing servers will be run on Amazon Elastic Compute Cloud (EC2) instances. The processing queue will be implemented using Amazon Simple Queue Service (SQS). Amazon Simple Storage Service (S3) will be used for storing content and configuration files, while Amazon SimpleDB will be used as the database for user data. The load balancer will be implemented using Amazon Elastic Load Balancing. Figure 3.4 shows the architecture of the various AWS components.

The web server and processing server instances will be split across several EC2 Availability Zones. These Availability Zones are designed by Amazon to ensure that any

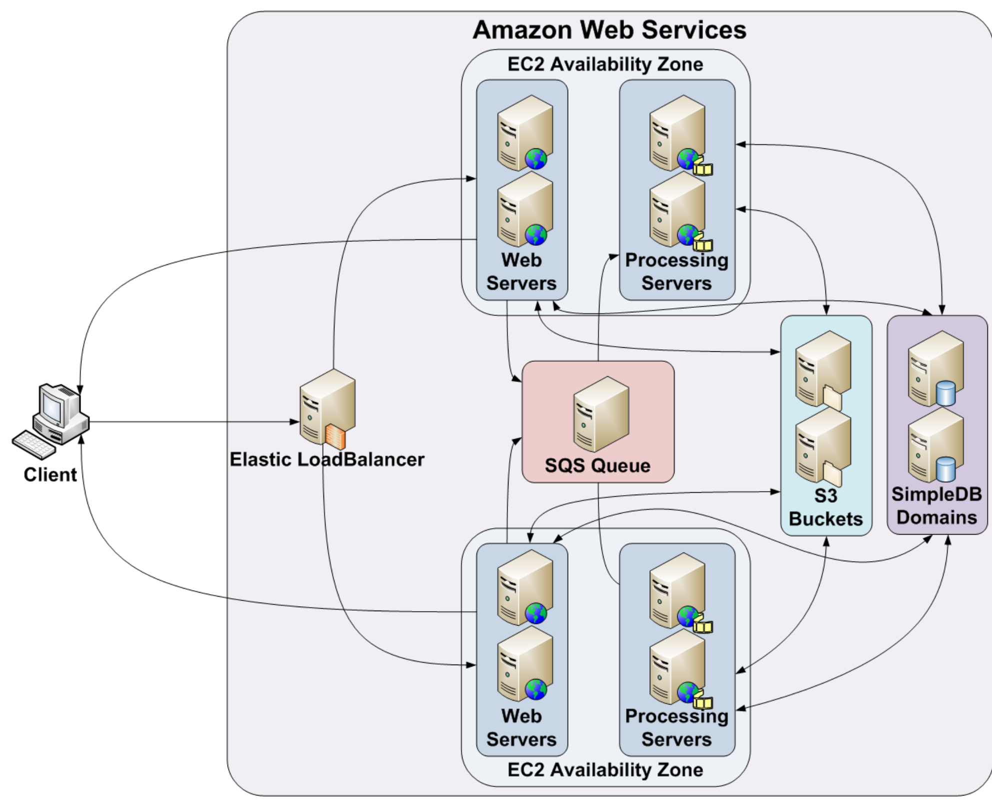


Figure 3.4: Network Architecture of AWS Components

network problems within the zone are restricted to that zone alone. The instances will still be able to communicate across the different Availability Zones.

All of the web server instances will be assigned to the Elastic LoadBalancer, which will have its DNS name tied to the domain name for the web application. The LoadBalancer will ensure that no one web server instance will become overloaded with traffic when other instances could handle that load.

In each Availability Zone each group of web server instances will be assigned to a single Auto Scaling Group. The Auto Scaling Group will control the number of web server instances running in that Availability Zone at any given time. Triggers will be assigned to the Auto Scaling Groups, using metrics from Amazon Cloud Watch. Once the resources of the group are being utilised to a certain degree Auto Scaling will automatically add more server instances or remove existing ones. Groups of processing server instances will be set up similarly to use Auto Scaling.

Web Server Architecture

The web server instances will be launched by EC2 from an Amazon Machine Image (AMI). This AMI will use Debian 5.0 as the operating system. It will run Apache Web Server

and PHP. The web front-end will be built into the AMI as once the front end is fully implemented it will be as static a component as Apache or PHP. The stack of the web server can be seen in figure 3.5

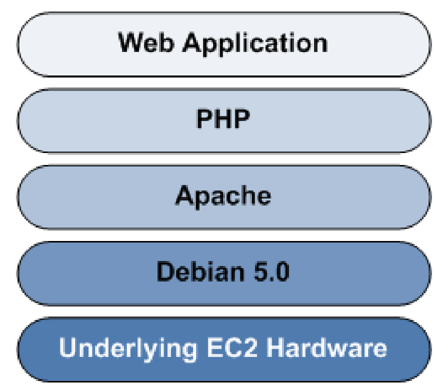


Figure 3.5: Web Server Software Stack

The web servers will have traffic directed to them from the various clients by the Elastic Load Balancer. The web servers will serve web pages to various clients and will receive content and configuration files from these clients, which they will then upload to S3 via the REST API. They will also insert metadata for content and configuration files into SimpleDB.

The web servers will communicate with the processing servers via the SQS queue. They will create messages to put on the queue containing information on jobs that the processing servers must complete.

Processing Server Architecture

The processing server instances will be launched by EC2 from a separate AMI to the web servers. This AMI will again use Debian 5.0 as the operating system, however it will have no need to run a web server such as Apache.

The main function of the processing server will be to wait to read messages off the SQS queue and when it successfully reads a message to perform the appropriate transform on the appropriate piece of content. Some abstract pseudo-code is presented for the processing server below to describe its overall activities.

```
while (true) {
   if (message on queue) {
      retrieve content from S3
      transform content
      upload new content to S3
   } else {
```

```
    backoff for message
  }
}
```

Chapter 4

Implementation

The web application implemented is called Foundry and is capable of accepting UGC in the form of images (JPEGs, PNGs, etc.). It can process content with a wide variety of transforms, in whatever combinations and order the user wants. Users can then review their content, both original and transformed, within galleries via their web browser.

4.1 Storage

Persistent storage for the application has been implemented using Amazon's Simple Storage Service (S3) and SimpleDB. No data is stored persistently on the web and processing server instances themselves. Amazon's Simple Queue Service (SQS) is used as the medium for the web servers communicating with the processing servers.

4.1.1 APIs

The web and processing servers communicate with S3, SimpleDB and SQS using the REST APIs provided by Amazon. These APIs use standard HTTP headers and additional headers added by Amazon to make requests to and receive responses from AWS services.

Amazon provides PHP libraries for using the REST APIs for SimpleDB and SQS, which makes using the API simpler than constructing the headers manually. At this time Amazon does not provide a PHP wrapper for the S3 REST API.

4.1.2 S3

A number of S3 buckets have been set up to store content (original, transformed and thumbnailed), configuration files and some system-related data. Each bucket on S3 must

have a globally unique name, so each bucket name uses `foundry` as a prefix (`foundrycontent`, `foundryconfig`, etc.).

Object Keys

Within the content and configuration buckets, the keys for every file use a specific schema. Every key begins with the user's username followed by a forward slash e.g. `smithco/`. This creates a hierarchical directory-like structure within the buckets, so that all of a user's content or configuration files can be listed.

Following the username prefix, the keys for configuration files and originally uploaded content take the form `rand_timehash_filename.ext`. Here `rand` is a random number generated by PHP's `mt_rand()` function, `timehash` is an MD5 hash of the current time returned by PHP's `time()` function and `filename` and `ext` are the original filename and extension of the file as uploaded by the user.

The keys for transformed content are slightly different, taking the form `username/rand_timehash_originalcontentkey`. Here `username`, `rand` and `timehash` are as described above. The `originalcontentkey` is the key for the original content that the transform is derived from, with the username prefix removed.

For thumbnails of both original and transformed content, the keys take the form `username/tn_originalkey`. Here username is as described above, `tn` is simply the string "tn" and the `originalkey` is the key for the content the thumbnail is created from, with the username prefix removed.

An example of a content key is shown below. It's the key for the thumbnail of a piece of transformed content with the original filename IM125.jpg:

```
smithco/tn_1562510234_87382f9aa0aa41a0c53dc253eb856836_1111655965_
26799419637391bc9e33b50787720905_IM125.jpg
```

System-Related Buckets

In addition to the buckets for content and configuration files, there are buckets for storing data used exclusively by the application itself. The `foundryschema` bucket contains the XML Schema file for the application. Changes to the schema can be made by replacing the XSD file sitting in the bucket with an updated version.

There is another bucket set up called `foundrylogs`. This bucket stores log files from the processing server, which are uploaded by each processing server at certain intervals. The key for every log file takes the form `instance_timehash_rand_log.txt`. Here `timehash` and `rand` are as described above, `log.txt` is the string as shown and `instance`

is a random number generated by PHP's `mt_rand()` function when a processing server instance boots and kept for the lifespan of that instance.

4.1.3 SimpleDB

A number of SimpleDB domains have been set up to store metadata about content and configuration files, as well user-related data and some system data.

For content, configuration file and user metadata, multiple domains are set up. There are four domains for configuration metadata, four domains for user metadata and eight domains for content metadata. Splitting the data across multiple domains allows faster reads and writes to each domain.

For every user all of their content metadata is within a single domain, as is all of their configuration metadata and user-related data. The domains that their data will be placed in are decided when the user registers. The function to determine the domains for a user involves getting the MD5 hash of their username (which must be unique).

Content Metadata

For the content metadata, the location of the content on S3 is used as the key for the domains. Other attributes for the content domains include: the title and description for the content, which is set by the user; the "content group", which for original content is its location and for transformed content is the location of the original content; a thumbnailed attribute, set only when a thumbnail has been created for a given piece of content; the owner, which is the unique username of the user that uploaded the original content; transform metadata, which is only set for transformed content and is a description of the transforms applied to that content.

Configuration Metadata

For the configuration metadata, the location of the content on S3 is used as the key, similarly to the content metadata. Other attributes for the configuration domains include: the title for the configuration file, which is set by the user; the owner, which is the unique username of the user that uploaded the configuration file.

User Metadata

For the user metadata, the user's unique username is used as the key for the domains. The registration process ensures that the username is unique. Other attributes for the user domains include: the first name and surname of the user; the user's email address; the

user's password, stored as an MD5 hash of the password the user entered at registration; the content and configuration domains that the user's content and configuration metadata is stored in.

Transform Code Domain

In addition to the domains set up to handle user metadata, there is a domain that contains the code that allows the processing servers to determine the ImageMagick command to use for a particular job it receives. In this domain, the name of the transform is used as the key. The rest of the attributes of the domain are entitled code, code2, code3... etc. Since the value for each attribute-item pair in SimpleDB is limited to 1,024 bytes in size, more than one attribute-item pair may be needed to store the code for a given transform.

4.2 The Web Servers

The front-end of the application is based on a number of web servers, running as instances on Amazon's EC2. These web servers deal with registering and keeping track of users, handling content and configuration file uploads, allowing users to retrieve their content, allowing users to edit their content and configuration metadata and allowing users to remove content and configuration files from the application.

4.2.1 Initial Implementation

Before implementing the web server on EC2, a version was implemented on a dedicated server running a LAMP stack. Initially the storage layer for the web server used a local MySQL server for storing metadata and local file storage for storing content and configuration files. These were gradually replaced with calls to the S3 and SimpleDB API, before finally porting the code over to EC2.

4.2.2 AMI

The AMI for the web servers was created using a public AMI using Debian 5.0 (lenny) as the operating system. To customise an AMI, the user must first log into it as `root`.

Using this AMI as a base, PHP 5 and Apache2 were installed and configured for use, with PHP's maximum file size altered to accept files up to ten megabytes in size. The code for the web server was then uploaded to the web directory.

The altered AMI was then bundled to save these alterations and create the web server AMI. Using tools provided by Amazon, a "snapshot" is taken of the current state of the

system. This snapshot is then compressed, encrypted and split for uploading.

The web server AMI was uploaded to S3, into a bucket named `foundryamis`. When the application needs to scale, the AMI is retrieved from this bucket and launched as an EC2 instance.

4.2.3 Configuration Files

Configuration files for the application are written in XML and must conform to the application's XML Schema. The schema describes the names of transforms that are available to use in the application. Each transform has certain options associated with it. The schema describes the names of valid options for transforms and the range of valid values for each of these options.

An example XML Schema for the application can be found in Appendix A.

Describing Transforms

Each configuration file contains one or more groups. Each of these groups describes one or more transforms to be performed, what order to perform the transforms, what combinations of transforms to perform and whether or not only unique combinations of transforms should be transformed.

Every group has a type. If this type is set to `single` then each transform within the group will be performed once. If this type is set to `multiple` then there are other attributes that must be set. The `combinations` and `unique` attributes decide what combinations of transforms will be performed. `Combinations` describes the number of transforms from the group to perform e.g. `combinations` is set to two, so transforms from the group are carried out on the content in pairs.

The `unique` attribute decides whether only unique combinations of transforms will be carried out or all possible permutations e.g. if `unique` is set to true and `combinations` is set to two, the transform `rotate` followed by `black&white` will be performed, but `black&white` followed by `rotate` will not.

The order of the transforms within the group determines the order in which each transform will be performed on a piece of content.

Each transform within the groups describes the name of the transform to perform, the associated options and values for those options. If a user does not enter a particular option for a transform, a default value will be used by the processing server executing the transform.

An example of a transform within a configuration file can be seen below:

```
<transform>
   <transformName>rotate</transformName>
   <options>
      <option>
         <direction>clockwise</direction>
      </option>
      <option>
         <degree>180</degree>
      </option>
   </options>
</transform>
```

Uploading and Managing Configurations

The web server handles users uploading their configuration files and passing those files onto storage in S3. When a user wishes to upload a new configuration file they select the file from their own machine and give a name for the file. Each user is limited to having one hundred configuration files at most.

Once the file is uploaded the web server checks if the file is actually XML. If it is not the user is given an error message telling them this. If the file is XML, then the web server attempts to validate it against the XML Schema, which it retrieves from S3. If the configuration file doesn't validate, the user is given an error message telling them this.

If the file is both valid XML and conforming to the schema the web server will upload it to the `foundryconfig` bucket on S3. Metadata for the configuration file, such as the name entered by the user, will then be added to the user's configuration domain on SimpleDB.

The application provides a page for users to manage their configuration files, wherein the user can rename any of the files they have uploaded and can also delete any of them.

Default Configurations

In addition to the user's own configuration files, the application provides a number of default configuration files for use. When selecting a configuration to use when uploading content, the user's configuration files are listed first, followed by the default configurations.

These default configurations can be added, altered and removed rather easily. The list of default configurations that every user sees is simply the configuration files for a special user named `admin`. To change anything about the default files, one just has to log in as `admin` and manage the configuration files from there.

4.2.4 Creating Jobs

It is the web server's task to parse configuration files and create jobs for the processing server from them. These job messages are encoded in XML and sent to the queue for the application on SQS. An example of a job message can be seen below:

```
<job>
  <content>
    smithco/1111655965_26799419637391bc9e33b50787720905_IM125.jpg
  </content>
  <domain>content2</domain>
  <transforms>
    <transform>
      <name>flip</name>
      <options>
        <option>
          <axis>horizontal</axis>
        </option>
      </options>
    </transform>
  </transforms>
</job>
```

If the configuration file used for the content has a group using multiple combinations, the web server must work out which combinations of transforms from the group are to be performed and send out jobs to the queue for each of them. In addition to job messages created from all of transforms listed in the configuration file, the web server sends out one extra job message. This job is to create a thumbnail of the original piece of content uploaded. An example thumbnail job message can be seen below:

```
<job>
  <content>
    smithco/1111655965_26799419637391bc9e33b50787720905_IM125.jpg
  </content>
  <domain>content2</domain>
  <thumbnail />
</job>
```

Both the web servers and processing servers parse through the various XML files and messages they receive using PHP's in-built SimpleXML parser.

4.2.5 The User Interface

The user interface for a web application is quite important. A badly designed and hard to use interface will make getting users difficult and would mean that designing the application to be large-scale would all be for nothing.

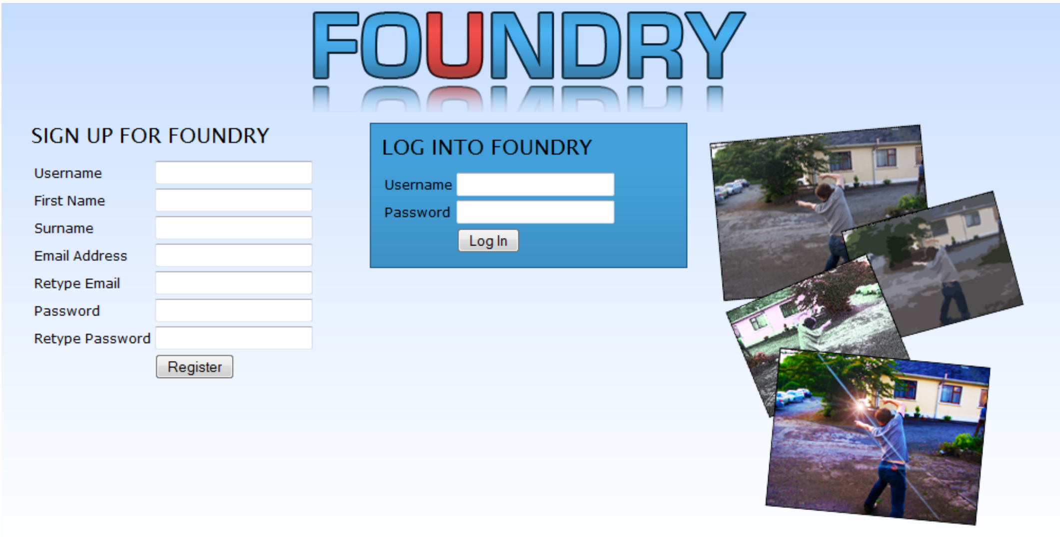


Figure 4.1: Registration and Login Page

Registration

All functions of the application are for registered users only. Visiting the front page of the application gives the user two choices: log in or register. The page can be seen in figure 4.1

The registration process involves the user selecting a unique username, a password and entering personal details such as their name. When the user registers the application checks to see if the username they have selected is unique. If it is already being used by another user then an error message is presented informing the registering user of this fact.

The user must enter their desired password twice and it is checked to ensure the user typed the same password twice. This password is then hashed using MD5 and sent to SimpleDB along with the rest of the user's details. It is at this stage that the application also determines which content and configuration domains to store the user's metadata in.

User Tracking

When a user logs in the application retrieves the MD5 hash of the password for the username they entered from SimpleDB. If that matches an MD5 hash of the password

they have entered the user is logged in. Otherwise an unspecific error is returned to the user telling them their login effort failed.

Once logged in the application keeps track of the user using cookies. Due to the user hopping between various web servers behind the load balancer PHP sessions cannot be used. Malevolent users are prevented from trying to pretend to be another user by a cookie known as the `userkey`. When the user logs in, the cookies, including the `userkey`, are set on their machine. The `userkey` is an MD5 hash of the unique username of the user combined with a static string. Whenever the application checks if the user is logged in, their `userkey` cookie is checked against a freshly generated key based on their username cookie.

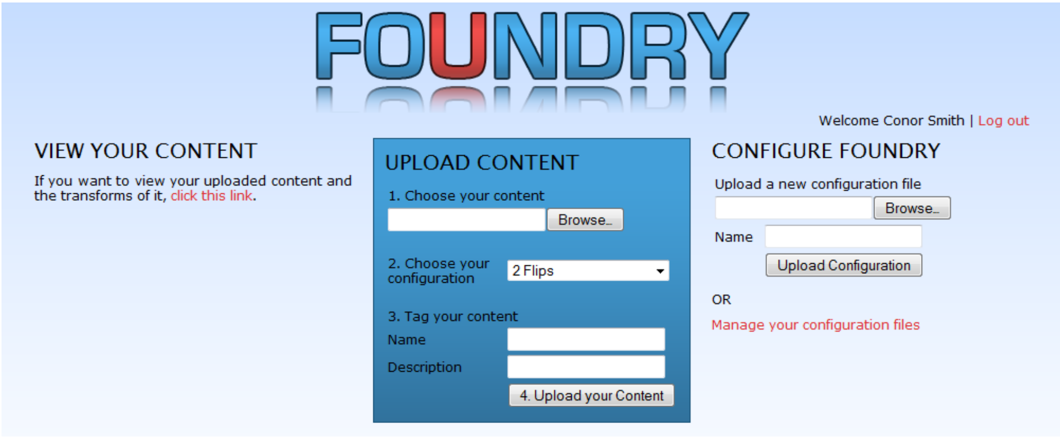


Figure 4.2: Front page when logged in

Uploading Content

When the user is logged in the front page of the application provides the user with the option to upload content, as shown in figure 4.2 The content uploading process is designed to be as simple as possible and is labelled as being four easy steps. The first step, labelled "Choose your content", is for the user to select a piece of content to upload from their machine. The content must be less than ten megabytes or the uploading process will fail, returning an error message to the user.

The second step, labelled "Choose your configuration", is for the user to select a configuration file from a drop down menu. The list is divided into two sections, with the first containing the user's own configuration files and the second containing the application's default configuration files.

The third step, labelled "Tag your content", encourages users to give a name and description for the piece of content they are uploading. This metadata will be applied to the original content and also all transforms of it. The fourth step, labelled "Upload your

content", is simply an HTML form submit button that the user presses to initiate the uploading process.

Figure 4.3: An example of a gallery page

Viewing Content and Transforms

Users can view the content they've uploaded using the application's gallery pages. There are two types of gallery pages: the main gallery, of which there is only one for each user; transform galleries, of which there is one for every piece of content each user uploads. Figure 4.3 shows an example of the main gallery and figure 4.4 shows an example of a transform gallery. All galleries are paginated, showing at most twenty-four pieces of content per page.

The main gallery shows thumbnails of all of the user's originally uploaded content. Under each thumbnail the name and description of the content is displayed and under this lies a list of options:

View: This option is a link to the piece of content on S3.

View Transforms: This option opens up the transform gallery for the piece of content.

Edit: This option allows the user to alter the name and description of the piece of content.

Delete: This option allows the user to delete the piece of content and all of its transforms. It prompts the user to ensure they have intended to click this option.

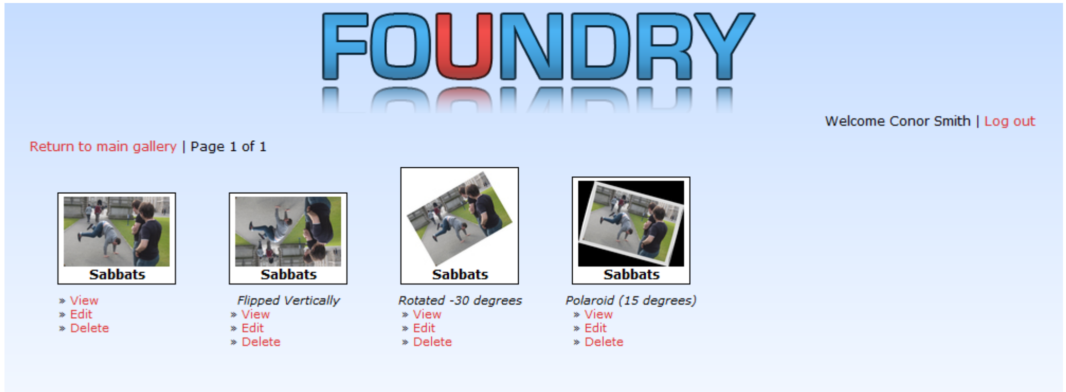


Figure 4.4: An example of a transform gallery page

The transform galleries are quite similar to the main gallery. The original piece of content is shown (thumbnailed) followed by all of the transforms for that content. If a particular transform has yet to be carried out on the content, then the transformed content will simply not be displayed.

Options for the transform galleries are similar to those for the main gallery, without the "View Transforms" option. When the delete option is clicked for a transformed piece of content, then only that single piece of content will be deleted, but if it is clicked on the original then all of the transforms will also be deleted.

If a piece of content has not had a thumbnail processed when a user tries to view it, the application will show a string stating "Thumbnail not available yet".

4.3 The Processing Servers

The processing servers run as instances on EC2. It is their job to read job messages off the queue and transform content as specified by the messages.

The code for the processing server runs in an infinite loop. It checks the queue for messages and, if it can retrieve one, processes the job and transforms the specified content. If there are no messages in the queue the application will sleep momentarily and check again. The length of time the application sleeps for increases as the number of times it has checked the queue and received no messages increases. This prevents the application from making unnecessary requests to the queue, which is useful as Amazon charge per request.

4.3.1 AMI

The AMI for the processing servers was based off the AMI for the web servers. In addition to using Debian 5.0 with PHP 5 installed, the processing server AMI has ImageMagick installed and configured to use as the program for performing the actual image transforms.

The code for the web server was removed and the code for the processing server added. The AMI was also altered to automatically run the processing script when it boots up.

4.3.2 Processing Jobs

The processing server reads jobs off the SQS queue one at a time. When the server reads a message it sets the message's visibility timeout to three minutes, meaning that no other processing server can read that message again for another three minutes. This gives the server enough time to process the message and perform the transforms specified, but is short enough to allow other servers to take on the job specified by the message if the first processing server terminates for some reason.

The job message specifies which piece of content to transform (giving its location on S3) and where the metadata for that content is stored (the SimpleDB domain). Using this information the processing server is able to retrieve the content and store it locally in a temporary file.

If the message is for a thumbnail job, then the processing server uses the ImageMagick `-thumbnail` option to create a thumbnail one hundred pixels wide. This thumbnail is then uploaded to S3 and the `thumbnailed` attribute for the original content is set.

If the message is for a transform job, then the processing server uses the name of the transforms to retrieve the transform code from the `transformcode` domain in SimpleDB. The code for each transform is written in PHP and executed using PHP's `eval()` function. It specifies how to create an ImageMagick command from the options for the transform given in the job message.

Once the processing server has put together the ImageMagick command it executes it directly using PHP's `shell_exec()` function. The ImageMagick command being used is called `mogrify`, which will perform a series of transforms on a given image and write the new image over the old image. A typical command derived from a job message may look like this:

```
mogrify -flip -rotate 60 temp/tempimg
```

Here, `-flip` and `-rotate 60` are two transforms being performed on the image `temp/tempimg`. First the image is flipped vertically by the `-flip` option, and then it is rotated sixty degrees clockwise by the `-rotate` option.

If the content is successfully transformed it is uploaded back to S3. The processing server then thumbnails the transformed content and uploads that thumbnail to S3. Following this metadata for the transformed content is uploaded to the correct content domain in SimpleDB.

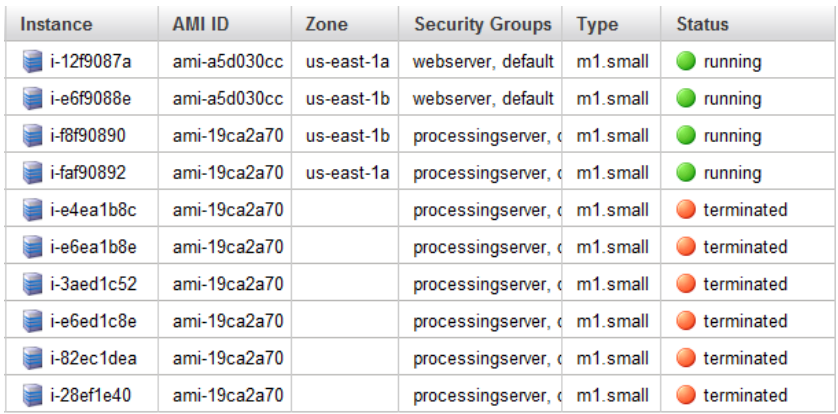
Once all of this has been done correctly the processing server deletes the message from the SQS queue, meaning that no other processing servers can read it and try to transform the original content again.

Logging

The processing servers keep a log of their activities as they run. Debugging information and other messages are written to a temporary file on each instance as the server's script runs. After every fifty transforms that the server performs this file is uploaded to the `foundrylogs` bucket on S3.

4.4 Launching the Application

Launching the application isn't simply pushing a button that says go. Using the API tools provided by Amazon, a load balancer and a number of launch configurations and auto scaling groups must be created to handle scaling the application. The number of instances running and their status can be viewed at any time using the AWS Management Console, shown in figure 4.5.



Instance	AMI ID	Zone	Security Groups	Type	Status
i-12f9087a	ami-a5d030cc	us-east-1a	webserver, default	m1.small	running
i-e6f9088e	ami-a5d030cc	us-east-1b	webserver, default	m1.small	running
i-f8f90890	ami-19ca2a70	us-east-1b	processingserver, (	m1.small	running
i-faf90892	ami-19ca2a70	us-east-1a	processingserver, (	m1.small	running
i-e4ea1b8c	ami-19ca2a70		processingserver, (	m1.small	terminated
i-e6ea1b8e	ami-19ca2a70		processingserver, (	m1.small	terminated
i-3aed1c52	ami-19ca2a70		processingserver, (	m1.small	terminated
i-e6ed1c8e	ami-19ca2a70		processingserver, (	m1.small	terminated
i-82ec1dea	ami-19ca2a70		processingserver, (	m1.small	terminated
i-28ef1e40	ami-19ca2a70		processingserver, (	m1.small	terminated

Figure 4.5: The AWS Management Console

4.4.1 Load Balancer

One load balancer is used in the application (`FoundryLoadBalancer`), which sits in front of the web servers and balances traffic across them. The load balancer listens for HTTP traffic on port 80 and sends traffic to the web servers on port 80. The load balancer is set up to work with instances in two availability zones.

4.4.2 Launch Configurations

Two launch configurations are used in the application, one for web servers (`FoundryWebLC`) and one for processing servers (`FoundryProcLC`). Each launch configuration has the ID of an AMI attached, as well as a list of security groups for the instances launching. Both types of server use the default security group, along with a specific security group set up for each type of server to allow the necessary traffic to get through. Both launch configurations are also set to use a small instance type for EC2.

4.4.3 Auto Scaling Groups

Four auto scaling groups are used in the application, two for web servers each in different availability zones (`FoundryWebASG1` and `FoundryWebASG2`) and two for processing servers also each in different availability zones (`FoundryProcASG1` and `FoundryProcASG2`). Each auto scaling group is tied to a launch configuration and has a minimum and maximum size. The minimum size for all groups is one and the maximum is one hundred.

The two web auto scaling groups are also tied to the `FoundryLoadBalancer`.

4.4.4 Triggers

Four triggers are used in the application, each tied to a different auto scaling group. These triggers tell the auto scaling groups when to scale up and when to scale down.

The web server triggers take the average amount of network traffic in over two minutes. If this is bigger than ten megabytes the group will scale up and if this is less than one megabyte the group will scale down.

The processing server triggers take the average amount of processor utilization over five minutes. If this is greater than sixty percent the group will scale up and if this is less than thirty percent the group will scale down.

Chapter 5

Evaluation

This chapter will describe the tests carried out to evaluate Foundry's scalability and the analysis of those tests' results.

5.1 Testing

A number of tests were carried out on the processing servers to see how they handle scaling up and down based on various loads.

In all of the tests carried out, five 2.5 MB images were uploaded to Foundry with each image set to have the same six pairs of transforms applied to them (first one transform is applied, then another before the transformed content is uploaded to S3).

This gives a total of thirty transform operations that have to take place, with an additional five thumbnail operations for the original images. Each transform operation takes approximately thirty seconds for ImageMagick to carry out. Each processing server instance takes approximately two minutes to become fully operational after it is launched. This is the time it takes EC2 to retrieve the AMI for the instance from S3, allot resources for that instance and get the operating system up and running.

Processing Server Test 1

The settings of the auto scaling triggers for this test were as follows: A new instance is booted when CPU utilisation exceeds 60% on average over a 5 minute period; A running instance is terminated when CPU utilisation is lower than 30% on average over a 5 minute period.

When the test began there were two processing server instances running (one in each availability zone). The transform and thumbnail operations all completed after 13 minutes. The maximum number of instances that were running during the test was eight.

Time (minutes)	Action
0	Test begins
4	2 new processing instances launch (1 in each Zone)
6	New instances fully booted
9	2 new processing instances launch (1 in each Zone)
11	New instances fully booted
13	All transform operations finish
15	2 new processing instances launch (1 in each Zone)
17	New instances fully booted
21	2 processing instances terminated (1 in each Zone)
26	2 processing instances terminated (1 in each Zone)
31	2 processing instances terminated (1 in each Zone)

Table 5.1: Processing Server Test 1

Time (minutes)	Action
0	Test begins
2	2 new processing instances launch (1 in each Zone)
4	New instances fully booted
5	2 new processing instances launch (1 in each Zone)
7	New instances fully booted
9	2 new processing instances launch (1 in each Zone)
11	New instances fully booted
11	All transform operations finish
15	2 processing instances terminated (1 in each Zone)
17	2 processing instances terminated (1 in each Zone)
19	2 processing instances terminated (1 in each Zone)

Table 5.2: Processing Server Test 2

The results can be seen in table 5.1.

Processing Server Test 2

The settings of the auto scaling triggers for this test were as follows: A new instance is booted when CPU utilisation exceeds 60% on average over a 2 minute period; A running instance is terminated when CPU utilisation is lower than 30% on average over a 2 minute period.

When the test began there were two processing server instances running (one in each availability zone). The transform and thumbnail operations all completed after 11 minutes. The maximum number of instances that were running during this test was again eight. The results can be seen in table 5.2.

Time (minutes)	Action
0	Test begins
3	2 new processing instances launch (1 in each Zone)
5	New instances fully booted
8	2 new processing instances launch (1 in each Zone)
10	New instances fully booted
13	All transform operations finish
15	2 processing instances terminated (1 in each Zone)
21	2 processing instances terminated (1 in each Zone)

Table 5.3: Processing Server Test 3

Processing Server Test 3

The settings of the auto scaling triggers for this test were as follows: A new instance is booted when CPU utilisation exceeds 90% on average over a 5 minute period; A running instance is terminated when CPU utilisation is lower than 20% on average over a 5 minute period.

When the test began there were two processing server instances running (one in each availability zone). The transform and thumbnail operations all completed after 13 minutes. The maximum number of instances that were running during the test was six. The results can be seen in table 5.3.

5.2 Scalability Analysis

The scalability of Foundry has been analysed based on Henderson's definiton of scalability: that a scalable application is one that can handle usage increases, can handle data increases and is maintainable [1]. The following sections discuss the evaluation of the application based on these criteria.

5.2.1 Usage Increases

Foundry's ability to handle increases in its usage relies on the infrastructure and services Amazon have in place for use with their cloud computing services.

The application's availability is handled by a number of design decisions relating to the use of Amazon's various services. Using two separate Availability Zones simultaneously ensures that instances will always be running in at least one zone, as the odds of both zones succumbing to failures at the same time are insignificant. The auto scaling groups ensure that a minimum of one instance is running in each availability zone, so if an instance

terminates for whatever reason a replacement instance will be launched immediately and booted within roughly two minutes.

The only bottleneck regarding availability is the load balancer. If the load balancer forwarding traffic to the web servers were to crash for whatever reason the application would operate as usual, but would be unreachable. However, another load balancer can be launched quickly.

Foundry's ability to launch new instances when needed relies heavily on the settings of the triggers for the Auto Scaling service. Based on the results of the above tests, using a shorter period and lower threshold for the metrics of the triggers will generally result in more instances being launched more quickly. However, this can lead to an unnecessary amount of instances being launched in the case of a short spike in usage, as seen by additional instances still launching after all transforms have been completed in the first two tests on the processing servers.

Amazon's pricing model for EC2 instances is that a certain amount is charged for every hour an instance is running. However, if an instance runs for only five minutes, this is still billed as a full hour. Using short periods and low thresholds for the triggers can result in instances launching and terminating multiple times within an hour, increases the cost of the application. Using longer periods for triggers can reduce the cost of running the application on EC2.

One of the useful features of Auto Scaling triggers is that they can be updated on the fly without disrupting the application. This means that the triggers could be altered to account for any massive and unexpected usage increases, such as if the application were to be linked to from Slashdot or Digg.

5.2.2 Data Increases

Foundry uses S3 to store users' content and configuration files and SimpleDB to store metadata for data on S3 and other system data. These services ensure that storage capacity is not an issue as the application grows. Costs associated with data storage will increase linearly with the increase in data being stored. Similarly, costs will decrease linearly if data is removed from the application.

To ensure that performance is not impeded by data increases, multiple domains are used in SimpleDB for each type of data being stored. For instance, content metadata is spread across eight domains meaning that every select query made for content metadata only has to work over roughly an eighth of the total content metadata. The number of domains for each type of data can be increased at a later stage if performance starts to become hindered.

5.2.3 Maintainability and Functional Scalability

The software model used in the design of Foundry made the code for the application far more maintainable. For instance, if Amazon were to radically change their API for S3 only the storage layer of the web server and processing server would have to be altered to take into account these changes. There is no need to worry about missing any other calls to the API elsewhere in the code, it is all located within the `storage.php` file.

The application was also designing to be functionally scalable in some regards i.e. that additional functionality could be added to the application without having to rewrite the application's code.

One of the major aspects of the functional scalability of Foundry is the ability to add new types of transforms to the application without the code having to be altered. As has already been discussed the XML Schema is stored on S3 and the PHP code for performing transforms is stored in SimpleDB. Adding a new transform simply involves updating the XML Schema and inserting around twenty lines of code into the `transformcode` domain.

However, there are some aspects of the application which are not functionally scalable. If one were to want to add the ability for Foundry to transform media types other than images (e.g. videos) alterations would have to be made to several parts of the code, mainly in the processing server. New software for handling the actual transforms of the new media type would have to be installed on the processing server also.

Chapter 6

Conclusion

This chapter will summarise the web application created for this disseration, Foundry, and discuss future work that can be done with the application.

6.1 Foundry

Foundry is an easy-to-use web application that allows its users to upload images and specify various transforms to apply to these images. It takes advantage of being built on a cloud computing platform to process and store vast amounts of content just in case the user is interested in the outcome.

The application was successfully implemented using Amazon's Elastic Compute Cloud to run the application's web servers and processing servers, Amazon's S3 and SimpleDB services to store the application's data and Amazon's Simple Queue Service to allow communication between it's components.

Foundry's ability to scale up and down the number of instances it is running at any time relies on the Auto Scaling service provided for EC2. The tests run on the application show that the scaling behaviour of Foundry depends on the metrics being used by the auto scaling triggers, the period over which these metrics are added and the thresholds of when to scale up and scale down the number of instances running. The triggers that control when Foundry scales are dynamic, meaning that the application's scaling behaviour can be altered on the fly. The software model used in the design of the web server ensures that the application is maintainable.

6.2 Future Work

Foundry as it is currently is a fully functioning web application, but there are many ways in which to expand it.

Currently Foundry performs transforms on images that its users upload. This could be expanded to perform transforms on other types of media, such as video and audio. To implement video transforms a library such as OpenCV could be used. OpenCV is an open-source library for performing computer vision based tasks, including facial recognition and motion tracking [75]. To implement audio transforms a program such as SoX could be used. SoX, or Sound eXchange, is a command-line utility licenced under GPL for editing audio files and applying various filters to them [76].

Another angle to approach expanding the application from is to add functionality for developing a community within the application instead of expanding its technical functionality. The majority of successful large-scale web applications have some community or social networking aspect to them; even a basic commenting system can add a community aspect.

While it is currently completely possible for Foundry users to manually share their transformed content with others via URLs, in-built functionality that allows users to share content and comment on the content of others could make the application more popular and increase the amount of time users spend on Foundry.

Appendix A

Example XML Schema

The following is an example of an XML Schema for use with Foundry. This particular schema supports three different transforms: flipping images, rotating images and creating a polaroid-style border around images. The schema describes four different options for use with these transforms: the axis to flip an image on, the direction to rotate an image, the degree to which an image is rotated and an option for applying rotations only when a certain condition is met. The various ranges of values for these options are also described in the schema.

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="transformConfig">
<xs:complexType>
<xs:sequence>
<xs:element ref="group" maxOccurs="unbounded" minOccurs="1" />
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="group">
<xs:complexType>
<xs:sequence>
<xs:element ref="transform" maxOccurs="unbounded" minOccurs="1" />
</xs:sequence>
<xs:attribute name="type" type="xs:string" use="required" />
<xs:attribute name="combinations" type="xs:integer" use="optional" />
```

```
<xs:attribute name="unique" type="xs:boolean" use="optional" />
</xs:complexType>
</xs:element>

<xs:element name="transform">
<xs:complexType>
<xs:sequence>
<xs:element ref="transformName" maxOccurs="1" minOccurs="1" />
<xs:element ref="options" maxOccurs="unbounded" minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="transformName">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="flip" />
<xs:enumeration value="rotate" />
<xs:enumeration value="polaroid" />
</xs:restriction>
</xs:simpleType>
</xs:element>

<xs:element name="options">
<xs:complexType>
<xs:sequence>
<xs:element ref="option" maxOccurs="unbounded" minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="option">
<xs:complexType>
<xs:sequence>
<xs:element name="axis" type="axisType" maxOccurs="1" minOccurs="0" />
<xs:element name="direction" type="directionType" maxOccurs="1" minOccurs="0" />
<xs:element name="degree" type="degreeType" maxOccurs="1" minOccurs="0" />
<xs:element name="rotateOnCondition" type="rotateOnConditionType"
            maxOccurs="1" minOccurs="0" />
```

```
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:simpleType name="axisType">
<xs:restriction base="xs:string">
<xs:enumeration value="horizontal" />
<xs:enumeration value="vertical" />
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="directionType">
<xs:restriction base="xs:string">
<xs:enumeration value="clockwise" />
<xs:enumeration value="counter-clockwise" />
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="degreeType">
<xs:restriction base="xs:integer">
<xs:minInclusive value="0" />
<xs:maxInclusive value="359" />
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="rotateOnConditionType">
<xs:restriction base="xs:string">
<xs:enumeration value="width greater than height" />
<xs:enumeration value="width less than height" />
<xs:enumeration value="height greater than width" />
<xs:enumeration value="height less than width" />
</xs:restriction>
</xs:simpleType>

</xs:schema>
```

Bibliography

[1] Henderson, C. (2006) Building Scalable Web Sites, O'Reilly, May, 2006.

[2] Linux Information Project - Scalable Definition `http://www.linfo.org/scalable.html`

[3] Michael, M., Moreira, J.E., Shiloach, D., Wisniewski, R.W., (2007) Scale-up x Scale-out: A Case Study using Nutch/Lucene, Parallel and Distributed Processing Symposium, 2007.

[4] Dean, J., Ghemawat, S., (2008) MapReduce: Simplified Data Processing on Large Clusters, Communications of the ACM, January, 2008.

[5] Flickr `http://www.flickr.com`.

[6] WordPress `http://www.wordpress.com`.

[7] Web Hypertext Application Technology Working Group: Implementations in Web Browsers `http://wiki.whatwg.org/wiki/Implementations_in_Web_browsers`.

[8] World Wide Web Consortium: Browser Statistics `http://www.w3schools.com/browsers/browsers_stats.asp`.

[9] Smarty Template Engine `http://www.smarty.net/`.

[10] Perl Template Toolkit `http://www.template-toolkit.org/`.

[11] High-Scalability: Google Architecture `http://highscalability.com/google-architecture`.

[12] Barroso, L.A., Dean, J., Holzle, U., (2003) Web Search for a Planet: The Google Cluster Architecture, 2003.

[13] YouTube `http://www.youtube.com`.

[14] Facebook `http://www.facebook.com`.

[15] Weiss, A., (2007) Computing in the Clouds, netWorker, Volume 11, Issue 4, December 2007.

[16] Amazon Web Services http://aws.amazon.com

[17] Amazon Elastic Compute Cloud: Developer Guide, API Version 2009-04-04 http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-dg.pdf

[18] Amazon EC2: Instances http://aws.amazon.com/ec2/#instance

[19] Amazon CloudWatch: Developer Guide, API Version 2009-05-15 http://awsdocs.s3.amazonaws.com/AmazonCloudWatch/latest/acw-dg.pdf

[20] Amazon Auto Scaling: Developer Guide, API Version 2009-05-15 http://awsdocs.s3.amazonaws.com/AutoScaling/latest/as-dg.pdf

[21] Amazon Elastic Load Balancing: Developer Guide, API Version 2009-05-15 http://awsdocs.s3.amazonaws.com/ElasticLoadBalancing/latest/elb-dg.pdf

[22] Amazon Simple Storage Service: Developer Guide, API Version 2006-03-01 http://awsdocs.s3.amazonaws.com/S3/latest/s3-dg.pdf

[23] Amazon S3: WSDL http://doc.s3.amazonaws.com/2006-03-01/AmazonS3.wsdl

[24] Amazon SimpleDB: Developer Guide, API Version 2009-04-15 http://awsdocs.s3.amazonaws.com/SDB/latest/sdb-dg.pdf

[25] ISO 8601:2004 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40874

[26] Amazon Simple Queue Service: Developer Guide, API Version 2009-02-01 http://awsdocs.s3.amazonaws.com/SQS/latest/sqs-dg.pdf

[27] Amazon Elastic MapReduce: Developer Guide, API Version 2009-03-31 http://s3.amazonaws.com/awsdocs/ElasticMapReduce/latest/AWSElasticMapReduce-dg.pdf

[28] Google AppEngine http://code.google.com/appengine/

[29] Windows Azure Platform http://www.microsoft.com/azure

[30] Gibbs, K., (2008) Developers, start your engines, The Official Google Blog http://googleblog.blogspot.com/2008/04/developers-start-your-engines.html

[31] Google AppEngine Developer Guide: What is Google AppEngine? http://code.google.com/appengine/docs/whatisgoogleappengine.html

[32] Google AppEngine Developer Guide: Google Accounts API Overview http://code.google.com/appengine/docs/python/users/overview.html, http://code.google.com/appengine/docs/java/users/overview.html

[33] Google AppEngine Developer Guide: Mail API Overview http://code.google.com/appengine/docs/python/mail/overview.html, http://code.google.com/appengine/docs/java/mail/overview.html

[34] Schofield, J., (2008) Google angles for business users with 'platform as a service', The Guardian http://www.guardian.co.uk/technology/2008/apr/17/google.software

[35] Anderson, C., (2008) Annoucning AppDrop.com (host Google App Engine projects on EC2) http://jchrisa.net/drl/_design/sofa/_show/post/announcing_appdrop_com__host_go

[36] Chohan, N., Bunch, C., Pang. S., Krintz, C., Mostafa, N., Soman, S., Wolski, R. (2009) AppScale Design and Implementation, UCSB Technical Report Number 2009-02

[37] Google AppEngine Developer Guide: Quotas http://code.google.com/appengine/docs/quotas.html

[38] Google AppEngine Developer Guide: Datastore API Overview http://code.google.com/appengine/docs/java/datastore/overview.html, http://code.google.com/appengine/docs/python/datastore/overview.html

[39] What is Windows Azure Platform? http://www.microsoft.com/azure/whatisazure.mspx

[40] Chappell, D. (2008) Introducing the Azure Services Platform: An Early Look at Windows Azure, .NET Services, SQL Services, and Live Services, October, 2008.

[41] Amazon Web Services: Overview of Security Processes, September 2008 http://s3.amazonaws.com/aws_blog/AWS_Security_Whitepaper_2008_09.pdf

[42] Extensible Markup Language (XML) http://www.w3.org/XML/

[43] Extensible Markup Language (XML) 1.0 (Fifth Edition): Documents http://www.w3.org/TR/REC-xml/#sec-documents

[44] Lee, D., Chu, W. W., (2000) Comparative Analysis of Six XML Schema Languages, September, 2000. http://pike.psu.edu/publications/sigmod-record-00.pdf

[45] XML Schema http://www.w3.org/XML/Schema

[46] XML Schema Part 2: Datatypes Second Edition http://www.w3.org/TR/xmlschema-2/

[47] Vaquero, L. M., Rodero-Merino, L., Caceres, J., Lindner, M. (2009) A Break in the Clouds: Towards a Cloud Definition, ACM SIGCOMM Computer Communication Review, Volume 39, Number 1, January 2009

[48] Lenk, A., Klems, M., Nimis, J., Tai, S., Sandholm, T. (2009) What's Inside the Cloud? An Architectural Map of the Cloud Landscape

[49] EUCALYPTUS http://www.eucalyptus.com/open/

[50] Django http://www.djangoproject.com/

[51] Google Apps http://www.google.com/apps

[52] Microsoft Office Live http://www.officelive.com

[53] Salesforce.com http://www.salesforce.com

[54] WikipediaVision (beta) http://www.lkozma.net/wpv/index.html

[55] Amazon Mechanical Turk https://www.mturk.com/mturk/welcome

[56] Digg http://www.digg.com

[57] ImageMagick http://www.imagemagick.org

[58] ImageMagick: Formats http://www.imagemagick.org/script/formats.php

[59] ImageMagick: Application Program Interfaces http://www.imagemagick.org/script/api.php

[60] MagickWand for PHP http://www.magickwand.org/

[61] IMagick http://pecl.php.net/package/imagick

[62] phMagick http://www.francodacosta.com/blog/phmagick

[63] ImageMagick: Command-line Tools: Convert http://www.imagemagick.org/script/convert.php

[64] Avidan, S., Shamir, A. (2007) Seam-Carving for Content-Aware Image Resizing

[65] ImageMagick v6 Examples - Video Handling http://www.imagemagick.org/Usage/video/

[66] Ramakrishnan, R., Tomkins, A. (2007) Toward a PeopleWeb, Computer, vol. 40, no. 8, pp. 63-72, Aug. 2007

[67] Wikipedia: Statistics http://en.wikipedia.org/wiki/Special:Statistics

[68] YouTube Fact Sheet http://www.youtube.com/t/fact_sheet

[69] Cha, M., Kwak, H., Rodriguez, P., Ahn, Y., Moon, S. (2007) I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System, Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, pp. 1-14, 2007

[70] Holt, B., Lynn, H. R., Sowers, M. (2007) Analysis of Copyrighted Videos on YouTube.com http://www.vidmeter.com/i/vidmeter_copyright_report.pdf

[71] Adobe Photoshop CS4 http://www.adobe.com/products/photoshop/photoshop/

[72] GIMP - The GNU Image Manipulation Project http://www.gimp.org/

[73] Adobe Premiere Pro CS4 http://www.adobe.com/products/premiere/

[74] Final Cut Pro 7 http://www.apple.com/finalcutstudio/finalcutpro/

[75] OpenCV - Wiki http://opencv.willowgarage.com/wiki/

[76] Sound eXchange http://sox.sourceforge.net/