A UNIFIED RESOURCE PLATFORM FOR THE RAPID DEVELOPMENT OF SCALABLE WEB APPLICATIONS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Russell Palmiter

January 2009

COMMITTEE MEMBERSHIP

TITLE:	A UNIFIED RESOURCE PLATFORM FOR THE RAPID DEVELOPMENT OF SCALABLE WEB APPLICATIONS
AUTHOR:	Russell Palmiter
DATE SUBMITTED:	February 2009
COMMITTEE CHAIR:	Michael Haungs
COMMITTEE MEMBER:	Aaron Keen
COMMITTEE MEMBER:	Alex Dekhtyar

A UNIFIED RESOURCE PLATFORM FOR THE RAPID DEVELOPMENT OF SCALABLE WEB APPLICATIONS

Russell Palmiter

Abstract

This dissertation presents Web Utility Kit (WUT): a platform that helps to simplify the process of creating web applications. It addresses the need to simplify the web development process through the creation of a hosted service that provides access to a unified set of resources. The resources are made available through a variety of protocols and formats to help simplify their consumption. It also provides a uniform model across all of its resources making multi-resource development an easier and more familiar task. WUT saves the time and cost associated with deployment, maintenance, and hosting of the hardware and software in which resources depend. It has a relatively low overhead averaging 123 ms per request and has been shown capable of linear scaling with each application server capable of handling 120+ requests per minute. This important property of being able to scale to developer's needs can help to eliminate the expensive scaling process. Initial users of the platform have found it to be extremely useful and have paved the way for future improvements.

TABLE OF CONTENTS

1 INTRODUCTION

This chapter introduces the motivation for a unified resource platform, discusses how people are currently solving this problem, and then addresses our solution and what we've uniquely done to contribute to this problem domain. After introducing the problem solved, we discuss the organization of the rest of this dissertation.

1.1 Next Generation Web Applications

As the web evolves, the use of web applications continues to grow [1]. Businesses and consumers across the world are increasingly moving to the web as a platform for running their applications and interacting with their data [1]. It is said that web application development is growing faster than ever before and we can expect that trend to continue [2]. Due to the client-server nature of the web and the ubiquity of access that it brings, web applications are now supporting more users than ever before thought imaginable [2]. Increased uses and usage brings increased need to perform more tasks and support more users than applications of the past. Despite these trends the tools, languages, and services that are currently prevalent in web development fail to provide adequate support. Popular frameworks today fail to make access to common resources simple and likewise fail to scale to the demand of today's web application needs.

1.2 Solution

Web Services and Service Oriented Architectures (SOAs) are becoming a common solution to this problem. They provide developers the ability to consume a resource without having to build, deploy, and maintain it. Unfortunately, most of the

services available to developers today fail to standardize on protocol, format, and data model and are only sometimes capable of providing the scale developers need. This can create large amounts of work for developers who need to consume and integrate these services, defeating the original intent of services--to make development faster and easier. What is needed is a unified collection of services that all share the same protocol, format, and data model, all while being capable of scaling to meet the needs of modern web applications.

1.3 Contribution

The solution we've developed is a hosted service that provides developers the ability to consume the resources they need in a uniform manner without having to worry about the language, protocol, data format, or scale of the system they're building. The system is a hosted service, which prevents developers from the need to deploy and maintain the various resources it provides. It provides multiple protocols and data formats so that developers can use the protocol, format, and the respective tools in which they are most comfortable. It also standardizes on a common data model to increase familiarity across the different resources offered. In addition, it is built using a combination of load balancing and horizontal partitioning to achieve the scale necessary by large web applications.

1.4 Organization

This dissertation is broken up into nine chapters. The chapters are organized as follows:

- Chapter 2 provides background knowledge and vocabulary that will be necessary to understanding the unique solution we built to solve this problem. This will

include discussion of various web architectures, communication techniques, data formats, partitioning techniques, redundancy techniques, and finally a review of cloud computing.

- Chapter 3 describes the platform we've created and the benefits gained from using it to build web applications. Included will be a description of the protocols, formats, model, and resources it provides. Benefits discussed will include increase productivity, reduced maintenance, and transparent scalability.
- Chapter 4 discusses the design of the service starting with the overall architecture and then discusses in more detail how the protocols, formats, model, and individual resources are designed.
- Chapter 5 looks at an example application built using the platform and discusses the various elements involved in building WUT based applications. The example makes use of several resources while standardizing on a client-side language, protocol, and data format as to exemplify typical use.
- Chapter 6 is devoted to the analysis of the platform including performance, scalability, and usability. Performance comparisons are made to comparable services, scalability is measured across a varying numbers of servers, and usability feedback is gathered from a focus group of initial users.
- Chapter 7 presents several pieces of related work and discusses their functionality including similarities and differences to that of WUT. Related work includes a python sharding library, a public distributed hash table, and python based hosted application platform.

- Chapter 8 reviews the project and makes conclusions on its practicality and usefulness as a web development platform.
- Chapter 9 discusses future work and direction for the project including additional services, protocols, security, and other features that would benefit the platform through its inclusion in future releases.

2 BACKGROUND

In this chapter we will cover some of the knowledge and vocabulary concerning the techniques that were employed in the development of this dissertation. This includes topics like web application development, web architectures, redundancy, partitioning, and cloud computing. Although all of the material presented in this chapter pertains to web application development, most web applications will selectively employ the techniques presented. Techniques employed depend on the budget and usage of the application.

2.1 Web Application Development

In this section we will discuss how the web is evolving into a popular platform for hosting information and the affects that's had on the web including emergence of web applications, web services, and service oriented architectures.

2.1.1 Increased Popularity of Web Applications

The web is changing fast and almost if not all of this change has happened over our lifetime. There has been a shift from the static mosaic style web pages of the early nineties, whose main purpose was to serve up a variety of information, to web applications that provide a more interactive experience [2]. And it's not only just the content that's changing on the web. The type and number of people on the web has changed dramatically. What used to be only a handful of academics is now millions and counting of different consumers all around the world [3]. These continual changes cause

the websites we use to constantly need more content, new features, and support more users.

2.1.2 Web Applications

Web applications are the way we provide new features and constantly changing content. For the purpose of this paper we define a web application to be an application delivered over the HTTP protocol that provides context-specific information to the user. This definition does not apply to static web sites. We have excluded these because they are inherently simple to scale. Due to the stateless nature of static web sites they can be easily load balanced to provide scale outside of the scope in which we address here [4].

Note that for the purpose of this dissertation we will not distinguish between a web application and a web service, as they are architected in a nearly identical fashion. The only difference being that web services are typically intended for a machine rather than a human audience.

2.1.3 Web Services

A web service is a specific functionality delivered over the HTTP protocol for the purpose of providing a mechanism for another service or application to use [5]. They enable the componentization and reuse of traditional web applications [6]. Some of the popular technologies used to develop web services include: XML-RPC [7], SOAP [8], REST [9], and REST-RPC-Hybrids [10]. These technologies differ in both the semantics they use to represent the data they transfer and the methodology they use to make the data available (i.e. resource based vs. remote procedure call based). Web services often include a way in which one can discover what methods or resources are available as well as what parameters they expect for a particular service. Some of the popular technologies

for describing web services include: UDDI (Universal Description Discovery and Integration) [11], WSDL (Web Service Description Language) [12], and WADL (Web Application Description Language) [13].

2.1.4 Service Oriented Architecture

With the growing popularity of web services and the need for constantly changing and adaptable websites comes a methodology for creating reusable services that are built in a modular fashion and designed to be easily consumed [6]. The nature of these services allows them to be leveraged to create new and different applications that share some of the same algorithms and data, without the need to rewrite all of the pieces [6]. This modularization also allows for pieces to be easily swapped out, so that when one piece of technology is outdated, it can be easily replaced without affecting the others. Perhaps the greatest advantage of designing services in this way is that it allows for other business entities or even entirely different businesses to consume the knowledge and techniques of others with ease.

2.1.5 Cloud Computing

Cloud computing is possibly the next evolution in the way web applications get built and used. It includes cloud applications, cloud services, and cloud platforms; all of which comprise a style of computing in which dynamically scalable and often virtualized resources are provided over the Internet [14]. Users of cloud technology need not have "knowledge of, expertise in, or control over the technology" provided by the "cloud" that supports them [15]. The term cloud is a metaphor for the Internet and comes from the way the Internet is typically depicted in computer network diagrams. Like in computer network diagrams, the "cloud" provides an abstraction for the complex infrastructure it

conceals [14]. Typical cloud environments pool resources together (into one large cloud) and share those resources across many users. Users typically provision the resources they need and when they're done simply add those resources back into the available pool. This pooling of resources helps to increase utilization and thereby reduce overall cost. This possible savings in cost is not the primary advantage of cloud computing though. The primary advantage is still the lack of expertise and maintenance needed to support the technology. Not having to expend effort supporting the underlying technology allows companies to focus on their core competencies.

2.2 Web Architectures

This section presents several common web architectures. Web architectures are a system level pattern for the tiers and layers that make up the components of a web application. Web architectures help to achieve two major tasks: separate code into distinct areas of concern and provide scalability by allowing individual layers to grow and shrink [16]. We will start this section out by first discussing the separation of concern and scalability provided by these architectures and then discuss the various architectures that can be used to achieve these properties.

2.2.1 Separation of Concerns

When splitting logically related code into distinct parts (either layers or tiers[1]), developers gain many advantages collectively referred to as separation of concerns. An

[1] Note that a tier is different from a layer in that a tier refers to a physically separated entity whereas a layer is simply a logical structuring of related components that need not live on different hardware [17].

example of splitting logically related code into parts might be to run presentation related code on one tier hosted at the client's location, application related code on another tier hosted at the provider's location, and yet a third tier may handle the management of the data and be hosted at a partner's location [17].

This separation of concerns allows any of the tiers to be independently provisioned, deployed, replaced, or upgraded as requirements or technologies change. For example, a change of operating system in the presentation tier would only affect the user interface code [18]. Likewise replacing a failed component in the data tier wouldn't result in the need to change any other tier.

2.2.2 Scalability

Scalability refers to a system's ability to increase or decrease capacity as demand increases or decreases [16]. The need for scalability is derived from the changing nature of business that can cause the size of a problem to grow or shrink over time [19]. One might think that they can solve this potential problem by initially building a system large enough to handle they're largest possible user base (in essence over-estimating demand), and although often this is technologically feasible, infrequently does this make sense from a business perspective. Building an architecture that can serve millions of users when your current user base is only thousands requires unneeded effort and resources and therefore incurs extra expenses [19]. The goal is to build a system for appropriate to your current user base of thousands of users, but to architect it in such a fashion that scaling it up to serve millions will not require a costly redesign [19]. Scalability in this fashion meets the company's needs both presently and is adaptable enough to meet future needs as well. Scalability is typical described in two fashions: either vertical or horizontal.

Vertical scalability refers to a system's ability to handle more volume by adding abilities to you current machine(s) [20]. This is typically done by adding more storage, processors, or random access memory to build a bigger, faster, or just plain better machine. This is an expensive strategy, but often is the easiest strategy to implement. It also tends to be a very limited strategy, as a machine can only be so powerful at any particular point in time.

Horizontal scalability refers to a system's ability to handle more volume by adding more of the same hardware or software [21]. This approach is typically considered to be a more flexible form of scalability, since it is not inherently limited in the amount of volume it can handle

Separating logic into distinct tiers allows the architecture the added advantage of being able to scale any particular tier independent of others. For example, if there are not enough application servers to handle the incoming requests, one can scale the application logic tier independent of the presentation and data tier, as can be seen in the figure below.

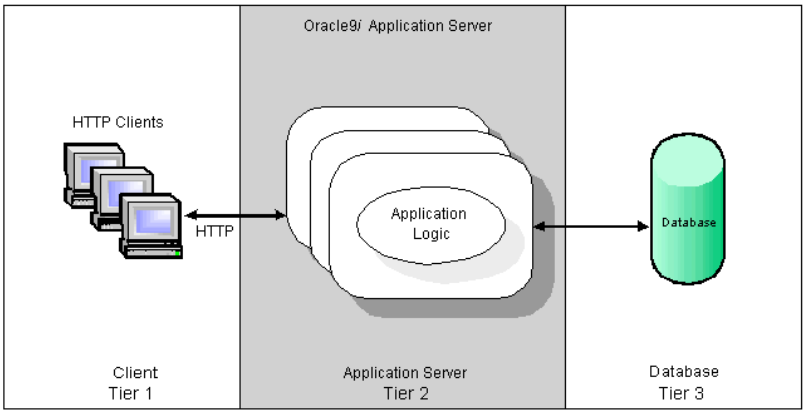

Figure 2.1 Multi-tier Application [cite]

2.2.3 Two-tier Web Architecture

A two-tier architecture, often simply referred to as a client-server architecture, describes the relationship between two computer programs in which one program makes requests for resources provided by another program. The two programs are typically named client and server. The client initiates requests and waits for replies [17]. Clients typically connect to only a small number of servers at one time [17]. The server on the other hand waits for and replies to requests from clients [17]. A diagram of this model can be seen in Figure 2.2. This simple model has become one of the central ideas of network computing [17], however, as applications grow to support more users they often outgrow this simple architecture.

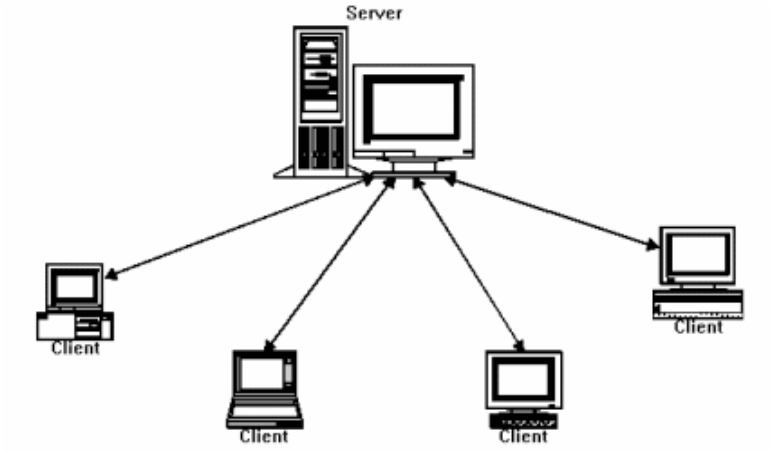

Figure 2.2 Two-tier web architecture

When developing a web application people often start with a simple two-tier architecture. They have a single server that contains both their web application and

potentially a database in an effort to keep the cost of hosting their application to a minimum; since all they need for a two-tier system is a single server.

2.2.4 Three-tier Architecture

The three-tiered web application architecture is perhaps the most typical and well known of web application architectures. The additional tier is typically a database management layer. This makes the three main architectural tiers are: presentation, application, and data [17]. A diagram of a three-tier architecture can be seen in the figure below.

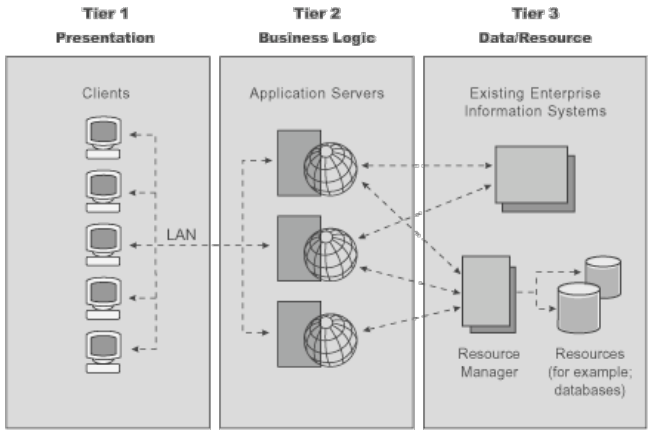

Figure 2.3 Three-tier Architecture [4]

The presentation tier is the topmost level of the application [17]. This tier is typically handled and rendered on the client-side by his or her browser. Layout and common user interaction are usually described in HTML and client side scripting languages like JavaScript are use to further engage the user and control complex interactions. It is also possible to use a plug-in based web technology like Flash, Flex, or Silverlight to accomplish more fine controlled interaction and presentation, such as interactive games, movies, and windowing.

The application tier, sometimes referred to as the logic tier, controls an application's functionality by performing detailed processing and manipulation of the data. This tier typical resides on the server-side, runs web server software, and is responsible for generating the presentation tier's content.

The data tier, sometimes referred to as the model tier, is responsible for storing and retrieving information. It keeps data independent from the application servers and business logic. It is typically implemented using a relational database management system. Giving data its own tier allows the rest of the application to scale more easily and typically helps performance. As web applications scale over time though, this is typically the layer that has the most difficulty scaling, as discussed below.

A fundamental rule in a three-tier architecture is the client tier never communicates directly with the data tier; or in other words all communication must pass through the application tier.

2.3 Load balancing

Load balancing provides a service from multiple separate systems where resources are effectively combined by distributing requests across these systems [22]. A diagram of a typical load balancer configuration can be seen below.

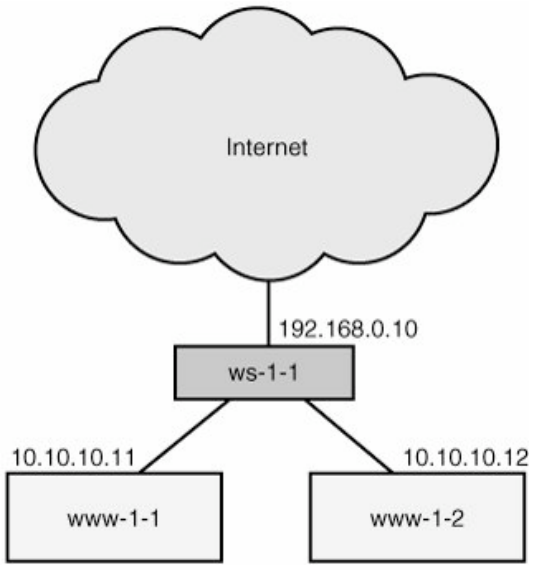

Figure 2.4 Load Balancer [5]

Another job typically performed by load balancers is the capability to run service checks against the servers to determine whether they are available to service requests. The service checks on these devices can range from simple ICMP pings to full HTTP requests with content validation. This means that if a machine fails, the load balancer will eliminate that machine from the eligible set of machines over which it distributes requests. As long as a single machine is alive and well in the pool of servers, the service as a whole will survive. This means the application servers are no longer a single point of failure. In other words, the service can survive N-1 application server failures. If one remains up, content continues to be severed [5].

Load balancing can be done in many different ways and implemented in either hardware or software. Some of the more popular approaches for distributing requests are described below.

2.3.1 Round robin

The round robin approach to load balancing involves a uniform rotation of requests between all the available servers. This approach unfortunately suffers from a queuing problem where servers that become overworked don't get the reprieve they require [5].

2.3.2 Available resources

Approaches that use available resources typically poll the servers they distribute requests to in order to retrieve periodic resource usage statistics. Due to the brevity of typical web transactions this approach often suffers from staleness issues [5].

2.3.3 Random

Randomized algorithms can pose an elegant solution to balancing requests that yield good probabilistic outcomes [5]. This approach, however, is typically complemented by resource-base approaches, as resource-based algorithms provide insight into ideal balancing and randomization can help to smooth out staleness issues [5].

2.3.4 Least connections

The least connections approach to load balancing distributes incoming requests to the server with the least number of open connections. This helps to distribute more requests to servers that are able to service more connections (accept, satisfy, and disconnect) [5]. In general, this is considered to be a good approach, however, due to the short-lived connections and highly parallel nature of web requests this method can suffer

from staleness. In addition to staleness, if multiple load balancers are used in parallel it can greatly affect this approach's ability to make good decisions [5].

2.3.5 Predictive

Predictive techniques are usually based on either a round robin, available resources, or least connections approach with some calculations to compensate for the information staleness issues induced by rapid transaction environments. These techniques can vary greatly, but a common solution is to use a statistical filter to determine what the load for a given server will likely be in the future and making distribution decisions based on that information [4].

2.3.6 Weighted

Weighted approaches attempt to compensate for known differences in server capabilities. They add a weighted factor to another approach that will cause the corresponding server to receive an adjusted amount of traffic. For example, if one server has twice as many core processors as another server it can be given a weight of 2, implying that it should receive twice as many requests. The problem with this approach is that it can be extremely difficult to accurately assign weights. It is unlikely that a dual core server will be able to handle twice as many requests as a single core server, so exactly how many more requests can it handle? Determining the answer to this likely takes a lot of trial and error [5]. In addition, changes to the system (either hardware or software) can cause the system to need re-weighting.

2.4 Redundancy

This section discusses a common technique to increase reliability and performance by keeping additional copies of the same data. We will discuss several common techniques that involve redundancy including caching, replication, and distributed databases.

2.4.1 Caching

Caching is a technique for duplicating data that is stored elsewhere or has been previously computed [23]. The purpose of caching data is to alleviate expensive access or computation of the original data. In essence, caches are a fast temporary storage location that helps to speed up an otherwise slow process of accessing or computing data. Caching systems vary widely in their implementation, approach, and applicability. Only proper analysis can determine if a caching is appropriate and if so which caching approach is best for specific situation [5].

2.4.2 Replication

Replication is a distributed technique where one replicates data across distinct severs to increasing the amount of read capability. This technique is most frequently use in association to a database, with many database vendors now supporting it out of the box.

This technique is most frequently used once the database becomes a bottleneck, which is likely to happen as you start adding application servers. It works by replicating data from a chosen master database to additional databases that act as slaves. All writes

are sent to the master database, which distributes this information out to its slaves, which in turn allows reads to be distributed across all available databases.

This technique becomes a bottleneck in your system when you have more writes then your databases can handle or you need access to your new data faster than the replication process can handle. In this case, people typically seek out a partitioning technique. More information about partitioning can be read in Section 2.5.

2.4.3 Distributed Caching

In a distributed cache, all the participating nodes share information in a common cache. This means when a particular node computes and stores results in its cache, other nodes will benefit. This makes caching a collective effort, rather than an individual one.

Distributed caches deal with data distribution in different ways, although almost all distributed caches store data in more than one place. Frequently, all of the data is stored at every node, although in more sophisticated systems the data may be replicated only on a certain number of peers and moved around as appropriate.

An effective distributed cache needs to do two things: pick the same node for a given key every time and spread keys evenly across all nodes. Although there can be many nuances to accomplishing these two tasks; the basic idea is that each node in the distributed cache is treated like a bucket in a hash table. Effectively used distributed cache's reduce contention on shared resources and as such increase both performance and scalability.

2.4.4 Distributed Database Management System

A distributed database management system is a database management system that provides transparent access to data that is stored on multiple computers across a network

[24]. This allows users to take advantage of the resources provided by multiple networked computers without any additional work. Typically distributed database management systems are intended to work in heterogeneous platforms.

Although the concept of distributed database management systems has been around for a long time, they are still not in widespread use. The reason for this is that algorithms and techniques used to manage traditional relational data add an unacceptable amount of overhead across multiple machines. For this reason most production environments favor other data distribution techniques.

2.5 Partitioning

In this section will we discuss an approach where data is divided amongst multiple machines in an attempt to increase the capacity and performance of data storage and access. A partition is a logical division of data's constituting elements into distinct independent parts [25]. The two primary types of partitioning will be discussed: vertical and horizontal.

2.5.1 Vertical Partitioning

Vertical partitioning is a partitioning technique that involves dividing elements into multiple sub-elements and storing the sub-elements in different distinct locations. This technique is often applied to databases whereby tables are divided by column into multiple tables with fewer columns. These tables are then typically stored on separate machines. Vertical partitioning divides data amongst machines and therefore lets each table store less data thereby increasing storage capacity and queries time. It does, however, make querying for complete elements a more difficult and possibly more costly task.

2.5.2 Horizontal Partitioning

Horizontal partitioning (often referred to as sharding) is a partitioning technique whereby individual elements of a collection are divided into distinct independent locations. Each independent location is referred to as a partition (or shard) and is likely located in a separate physical location. This technique is typically applied to databases where rows of a database table are held separately, rather than splitting by columns (as for normalization) [26]. The advantage is that the number of rows in each table are reduced, in turn reducing index size and improving search performance [26]. Also if the partitioning is based on some real-world aspect of the data (e.g. European customers vs. American customers) then it may be possible to infer the appropriate partition membership automatically, allowing one to only query the single relevant partition [26]. Although the premise of horizontal partitioning is relatively simple, it can be a difficult system to maintain. It is for this reason that is currently being employed mostly by large enterprises with the ability to build such a system, like that of Google [27], Yahoo [28], and Amazon [29].

There are many ways in which one can divide elements into separate partitions. The following are popular methods for horizontal partitioning of data.

2.5.2.1 Range Partitioning

This method selects a partition by determining if the partitioning key is inside a certain range. An example could be a partition for all rows where the column zipcode has a value between 93000 and 94000 [25].

2.5.2.2 List Partitioning

In this method a partition is assigned a list of values. If the partitioning key has one of these values, the partition is chosen. For example all rows where the column Country is either Iceland, Norway, Sweden, Finland, or Denmark could build a partition for the Nordic countries [25].

2.5.2.3 Hash Partitioning

In this method the value of a hash function determines membership in a partition. Assuming there are four partitions, the hash function could return a value from 0 to 3 [25].

2.5.2.4 Consistent Hashing

Consistent hashing is a scheme that provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped. By using consistent hashing, only K/n keys need to be remapped on average, where K is the number of keys, and n is the number of slots [26].

Consistent hashing was introduced in 1997 as a way of distributing requests among a changing population of web servers. Each slot is then represented by a node in a distributed system. The addition (joins) and removal (leaves/failures) of nodes only requires K/n items to be re-shuffled when the number of slots/nodes change. More recently it has been used to reduce the impact of partial system failures in large web applications as to allow for robust caches without incurring the system wide fallout of a failure [30] [31].

However, the most significant application of consistent hashing has been to form the foundation of distributed hash tables (DHTs). DHTs use consistent hashing to partition a keyspace among a distributed set of nodes, and additionally provide an overlay network which connects nodes such that the node responsible for any key can be efficiently located [31].

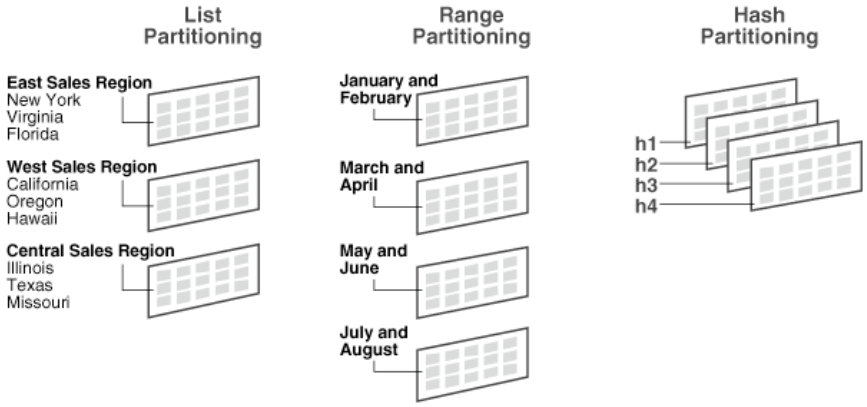

Figure 2.5: Different Partitioning Methods [4]

2.6 Summary

We have presented some state of the art techniques used to develop cloud applications. Web Utility Kit (WUT) utilizes several of these techniques internally and provides access to resources that also leverage these techniques in building their representative services. Internally, WUT is composed of a three-tiered architecture, uses a load balancer, and horizontally partitions data. All of the resources provided by WUT leverage a different selection of each techniques described, but in combination all of the techniques described are used.

3 WEB UTILITY KIT

This chapter describes the intent and benefits of using Web Utility Kit (WUT). It includes a description of the platform including its protocols, formats, data model, and resources. It also discusses how it helps to increase developer productivity, cut costs, and make overall development easier.

3.1 Overview

WUT is a platform intended for the rapid development of scalable web applications through an abstraction of resources that are made easily consumable through a variety of protocols and made available in multiple formats. WUT allows developers to leverage common features with very minimal work. It makes development easier and faster by hosting a service that provides developers the ability to consume resources in a uniform manner without having to worry about the language, protocol, data format, or scale of the system they're building. Because the service is hosted, developers don't need to worry about upfront capital costs, deployment, maintenance, or anything other than consumption of the service. This allows companies to concentrate on the development most closely related to their core competencies. In total, WUT provides four protocols, six formats, six models, and ten different resources. An overview of the service is depicted in the diagram below.

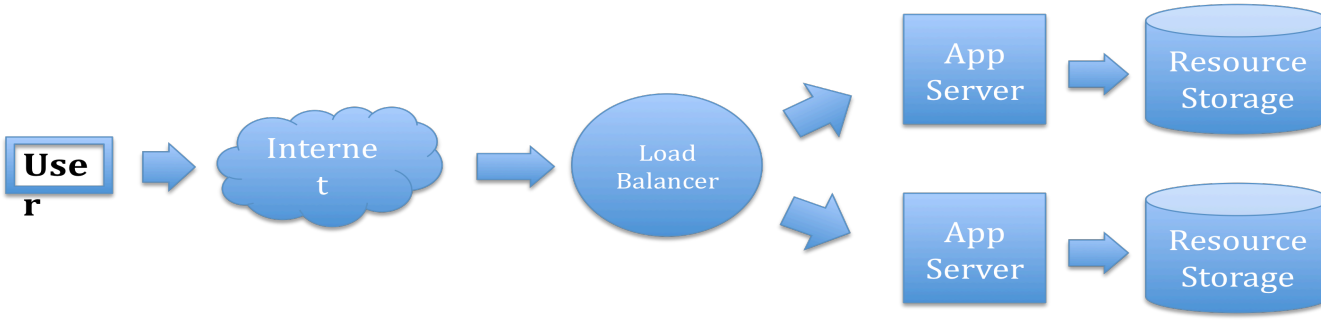

Figure 3.1: WUT Overview

<< fix wrapped text >>

3.2 Hosted Service

WUT is provided as a hosted service, which saves companies from large capital expenditures, difficult deployments, and costly on-going maintenance. Being a hosted service means that WUT has no components that need to be installed at a company's physical location and instead depends on the Internet as a means to access its resources. This trend in software development has been shown to be an effective way to share the cost of expensive hardware, software, and the personnel and experience necessary to maintain such hardware and software. Furthermore, having someone else concentrate on the support of resources for your application helps companies to concentrate on their core competencies and innovate in the areas most vital to their success.

Having resources already deployed and available allows a company to start development immediately, without the hassle of a potentially long and difficult deployment period. Deployment of a large web application can often involve many components including application servers, load balancers, caches, and database management systems. Deployment of these components can take days or even weeks and often requires expertise not already present within a company. To get this expertise,

companies typically hire consultants to come in and help with deployment. With a hosted service, this cost and delay are no longer necessary. The benefits become immediately apparent from the second developers first sit down to use the service.

Hosted services also avoid the large capital expenditures that can be associated with web application hardware and software. An enterprise web server costs anywhere from $10,000 to $50,000 [32]. Software can vary in price ranging from free to $40,000 per CPU [33]. A redundant two-node (4 CPU) cluster employing Oracle software for scalable data management currently costs $1,610,400 [34] (calculation can be found in Appendix C). As can be seen, these costs can be extremely high and an expensive upfront investment for a smaller company, sometimes not even affordable. And worse yet, it is frequent that companies purchase expensive software and hardware only to put it into production and learn that it doesn't provide exactly what they need and are then forced to replace the components they just purchased. This risk is eliminated through the use of a hosted service.

The on-going maintenance cost of running server software and hardware can be extremely large. Every component in the system has a unique set of maintenance needs that regularly need to be addressed. Servers need physical storage space, temperature control, bandwidth, monitoring, restarting, and upgrading of both software and hardware. Databases can require tuning, upgrading, and restarting if something is to go wrong. Caches can require flushing, restarting, or other such procedures if they are to get out of synchronization. All of these resources can change fail, change, or go away altogether, requiring you to replace or search out service from another provider of the resource. And as you application grows you'll have to add new servers, databases, caches, and other

components related to resources. This process can cost a tremendous amount of time and money, depending on a company's available resources and expertise.

3.3 Protocols

The protocol is the method of communication used to request and retrieve resources from WUT. Having multiple protocols available for consumption allows developers a wider range of languages in which to develop in. And since many languages have support for more than one of the protocols provided (if not all), it often allows developers to choice the protocol in which they are most familiar or simply just the one that best suites their needs. WUT supports the following protocols: REST, XML-RPC, REST-RPC-HYBRID, and SOAP. Each protocol accesses the same set of resources, the same model, and is capable of performing the same tasks, but each does so in a slightly different manner. Each protocol has its own advantages and disadvantages, but WUT leaves it up to the developer to decide which protocol is appropriate for their needs. Individual protocol advantages range from SOAP's variety of code generation tools to REST's simplicity and familiarity.

3.4 Formats

The format is the presentation of the data returned by the protocol. The data delivered by a particular protocol is often referred to as the protocol's payload. The payload is the result of a particular request to a resource. This is typically the data of importance to the developer. For a request to a photo resource, this payload will likely be that of the raw image itself. For a request to a web search resource the payload will likely be that of a list of corresponding websites. The format of the payload refers to the way in

which that data is described. Different resources may be limited in the formats they support. For a photo resource, the format of the image may be that of JPEG, GIF, or PNG. For a web search resource, the format may be that of XML, YAML, JSON, or HTML. Each particular format may have a set of associated languages, libraries, and tools that support it. For example XML has a number of schema, selection, and transformation languages associated with it all made available by numerous libraries supporting hundreds of languages. In addition to that there are tons of tools that help developers to view, write, and understand data described by the language. HTML on the other hand has a number of tools and languages available to assist in its presentation.

3.5 Model

The model is the data structure that stores the data to be returned by a particular request. WUT's model contains four basic types from which all other types must derive: scalar, list, map, and matrix. By using a common model for all of its resources, WUT is able to return data in a more limited and familiar representation. Even though there are only four base types, that does not limited the richness of WUT's object model. WUT has many derivative types that contribute to the richness of its data model, especially in its representation of scalar data. Currently WUT supports to following scalar types

Name	Description	Example
`NaturalNumber`	Positive numbers greater than 0	1
`WholeNumber`	Positive numbers including 0	0
`Integer`	Negative and positive whole numbers	-123
`Double`	Fractional Numbers	1.23423
`Boolean`	True or false	True

Id	Alphanumeric sequence	U012345
AutoId	Alphanumeric sequence set by WUT	U012346
Reference	Alphanumeric sequence that refers to another id	U012345
Date	A date of the format MM/DD/YYYY	1/12/2009
Time	Time of day	11:00 am
Timestamp	Time since epoch	10386005
Word	All alphanumeric characters limited in size to 100 characters	hello
Sentence	Sentence of text 256 characters or less	A cat ran fast.
Paragraph	Paragraph of text 1024 characters or less	A cat ran fast. Then it jumped. More words here.
String	No limit in size	
City	City	San Luis Obispo
State	State	California
Address	Combination of street address, city, state, and zip code	1 Grand Ave, San Luis Obispo, CA 93401
Zip Code	5 digit zip code	93401
Phone Number	Phone number with area code	(805) 756-1234
Email Address	Email address	russell@mailinator.com
Credit Card Number	Credit Card Number	123412342355
Binary Data	Binary Data	N/A

Table 3-1: WUT Scalar Types

3.6 Resources

This section outlines each of the ten resources currently available for consumption by developers. They serve a range of functions from returning photos to changing settings associated with a user's account. Although the resources currently available serve a wide range of needs, WUT was created with the intent of expanding its resource offering over time so it is expected that in the future the number of resources offered will grow.

Examples used in this section all use the REST protocol for requests and only display the payload of the responses. This is due to simplicity of presentation. More information on how other protocols make and respond to requests can be found in Section 4.4.

3.6.1 Settings Resource

The settings resource is a special resource used to configure the other resources. It controls a number of properties that are used throughout the system to control both how the system behaves and how it interacts with other systems. An example use of this resource can be found below.

```
http://www.webutilitykit.com/resource/admin/settings/name=email.pop3.us
ername&value=wutuser
```

Example 3.1 Settings Request

```
<results>
  <message>success</message>
</results>
```

Example 3.2 Settings Response

Resources that utilize settings managed by this resource will specify their dependence in the sections below.

3.6.2 Web Search

The web search resource provides the ability to gather information on web content. It allows users to retrieve links, titles, and short descriptions of web sites related to a given search term. An example request for websites related to dog training can be found below.

```
http://www.webutilitykit.com/resource/web/search/term=dog+training
```

Example 3.3 Web Search Request

```
<results>
  <map>
    <summary>
      Dedicated to purebred dogs and responsible dog ownership.
      Includes information on finding the right dog, training and
      obedience, and different breeds.
    </summary>
    <title>American Kennel Club (AKC)</title>
    <url>http://www.akc.org/</url>
  </map>
  <map>
    <summary>
      Find the answers to: Why do dogs howl? What makes a dog tick?
      What are the best training tips for dogs? Animal Planet&amp;apos;s
      Dog Guide covers dog anatomy, care, safety, ...
    </summary>
    <title>Animal Planet&amp;apos;s Dog Guide</title>
    <url>http://animal.discovery.com/guides/dogs/dogs.html</url>
  </map>
  <map>
  ...
</results>
```

Example 3.4 Web Search Response

3.6.3 Photos

The photos resource allows applications to store, retrieve, and search for images.

An example search request for photos of cats can be viewed below.

```
http://www.webutilitykit.com/resource/photos/search/term=cat
```

Example 3.5 Photo Search Request

```
<results>
  <map>
    <title>Puma at the Santago Leopard Project</title>
    <url>
      http://farm4.static.flickr.com/3546/3419008990_c04d1a8e2f.jpg
    </url>
  </map>
  <map>
    <title>Rosie and her babies...</title>
    <url>
      http://farm4.static.flickr.com/3660/3418991776_6b4f19d366.jpg
    </url>
  </map>
  <map>
    <title>Minnie</title>
    <url>
```

```
      http://farm4.static.flickr.com/3336/3418184933_26d5643be4.jpg
    </url>
  </map>
  ...
</results>
```

Example 3.6 Photo Search Response

To store photos, users must have the settings "photos.flickr.username" and "photos.flickr.password" set. To set these, the user would use the settings resource as described in Section 3.6.1.

3.6.4 Geocoding

Geocoding is the process of transforming an address or location into its respective latitude and longitude. The geocoding resource accepts the following location identifiers:

- city, state
- city, state, zip code
- zip code
- street, city, state
- street, city, state, zip code
- street, zip code

An example geocoding request for the address 1 Grand Ave, San Luis Obispo, CA 93410 can be seen in the example below.

```
http://www.webutilitykit.com/resources/geocode/encode/location=1+grand+
ave,san+luis+obispo,ca,93401
```

Example 3.7 Geocoding Request

```
<results>
  <state>CA</state>
  <address>1 Grand Ave</address>
  <zip>93405-9000</zip>
```

```
  <longitude>-120.652992</longitude>
  <country>US</country>
  <city>San Luis Obispo</city>
  <latitude>35.296107</latitude>
</results>
```

Example 3.8 Geocoding Response

Note that if the location specified identifies more than one location then multiple sets of latitude and longitude will be returned.

3.6.5 Payments

The payments resource allows developers to process credit cards. Processing credit cards requires a card number, CVV2, expiration date, card type, cardholder first name, cardholder last name, and billing address. This resource currently accepts visa, master card, discover, or american express cards. There is a 2% transaction fee for all processing requests. An example credit card transaction request can be seen below.

```
http://www.webutilitykit.com/resources/payments/process/name=Russell+Pa
lmiter&number=5375394765940827&expiration=0909&cvv2=887
```

Example 3.9 Payment Request

```
<results>
  <message>success</message>
</results>
```

Example 3.10 Payment Response

Before using this resource, applications must have their payments settings set appropriately. We realize that at this time WUT exposes information over HTTP in plain-text, making sending sensitive information over this protocol a risk. As identified in Future Work, SSL could be used to provide secure communication over HTTPS to overcome this problem.

3.6.6 Relational Storage

The relational storage resource provides the ability to create, store, retrieve, and delete relational records. Records are referenced by a unique identifier automatically assigned to them on creation. The relational storage engine supports selection, projection, and ordering, but not joining of data (as joining of data can be difficult to do in a scalable manner). An example of storing a supplier record in a virtual table can be seen below.

```
http://www.webutilitykit.com/resources/relational/virtual/table=supplie
rs&data={name:fabricsrus,phone:1-408-923-2111,contact:mary+jones}
```

Example 3.11 Relational Request

```
<results>
  <message>success</message>
</results>
```

Example 3.12 Relational Response

3.6.7 Hash Storage

Hash storage provides a way for users to store and retrieve data based on a given key. Users are responsible for creating managing keys including insuring uniqueness and recalling the appropriate keys for their data. Keys are not limited in any way, so it is up to the application to choose keys appropriate to its use. An example storage of data can be found below.

```
http://www.webutilitykit.com/resources/associative/key=12345&data=savem
e
```

Example 3.13 Hash Request

```
<results>
  <message>success</message>
</results>
```

Example 3.14 Hash Response

3.6.8 Session Storage

Session storage works exactly like the hash storage resource except that users are not responsible for deleting the data. Session storage has a system wide time-to-live (TTL) which is guaranteed to be at least 30 minutes, after which point the session resource can delete data at will. This resource is intended for the temporary storage of data, like that of a user's session data.

```
http://www.webutilitykit.com/resources/associative/key=12345&data=ilive
for30minutes
```

Example 3.15 Session Request

```
<results>
  <message>success</message>
</results>
```

Example 3.16 Session Response

3.6.9 Email

The email resource sends email from a user specified account POP3 email account.

```
http://www.webutilitykit.com/resources/email/to=rpalmite%40calpoly.edu&
subject=email+request&body=thiscamefromwut
```

Example 3.17 Email Request

```
<results>
  <message>success</message>
</results>
```

Example 3.18 Email Response

POP3 account information can be edited using the settings resource.

3.6.10 Web Scraping

The web scraping resource allows the user to retrieve a whole or parts of a webpage. They can specify an XPATH selector and a HTTP address. If possible the HTTP page given will be converted to XHTML with the appropriate selector applied.

```
http://www.webutilitykit.com/resources/web/scrape/url=
http%3A%2F%2Fwww.google.com&filter=%2F%2Finput%5Bvalue%3D'Google+Search
'%5D
```

Example 3.19 Web Scrape Request

```
<results>
  <input id="btnG" type="submit" name="btnG" value="Google Search"/>
</results>
```

Example 3.20 Web Scrape Response

3.6.11 Document

The document resource allows the user to create, update, retrieve, and delete documents. The documents can be of any type, format, and size. An example document request can be seen below.

```
http://www.webutilitykit.com/resources/documents/hello.rtf
```

Example 3.21 Web Scrape Request

```
<results>
  <![CDATA[
     {\rtf1
     {\fonttbl
     {\f0 Arial;}
     {\f1 Times New Roman;}
     }
     \f0\fs60 Hello World\par
     \f1\fs20 This is a simple RTF document\par
     }
  ]]>
</results>
```

Example 3.22 Web Scrape Response

3.7 Scalability

Now that cloud computing has made access to hardware and infrastructure increasingly flexible, software that can automatically leverage that flexibility is of great asset. Currently, if users want to make use of cloud computing's vast scalability, they have only a few options. They can use one of the few services currently available [35-37], or they can develop their own homegrown solution.

By using WUT they can gain a language independent set of scalable services, scaling both in computing power and storage. Users need not worry about how performance is achieved, or what is actually being done, that is all abstracted away and place behind the safety of WUT. That in turn allows them to concentrate on what really matters to them, their end user functionality.

4 DESIGN

This chapter starts by discussing the overall design of Web Utility Kit (WUT) and then talks about how the protocols, formats, and model all work to support access to the resources. It then discusses the individual design of each resource. In the process of this discussion we will explain how WUT achieves various elements of performance and scalability.

4.1 Overall Design

WUT is designed as a multi-tier application composed of independent resources that each tie together by taking advantage of common protocols, formats, and data models. It uses a load balancer to distribute requests to individual application servers. The application servers contain the protocols, formatters, data model, and resources that are used to service requests. It is expected that stateful resources store their state outside of the application server for reasons that are explained in section 4.3. A diagram of WUT's multi-tier architecture can be seen below.

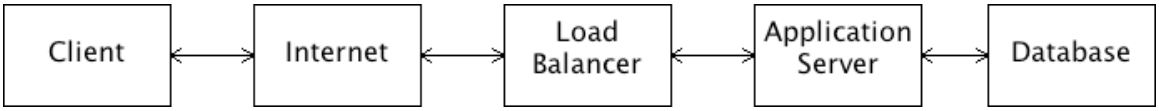

Figure 4.1: High Level Design Overview

4.2 Modularized Architecture

WUT is designed with a modularized architecture. This modularization between the components provides encapsulation that allows one module to freely change without affecting other modules. The four major modules in WUT are protocol, format, model, and resource. Information flows between the modules in a predefined manner. Requests are first handled by their respective protocol. The protocol then formally creates a `RequestContext` which contains all the information necessary to process the request. << generically its of the format in figure 4.X >>. This request context is then passed to the `ProcessingPipeline`, which utilizes the appropriate resource to fulfill the request. The resource is also in turn given the same `RequestContext`, which provides access to the request's payload if necessary for processing the request. The payload is made available in both formatted and model form. Having the payload in model format allows the resources to easily work with it and having the payload in already formatted allows resources to efficiently pass that data on where appropriate. A diagram of the interaction between modules can be viewed below, along with a diagram that shows particular components within the modules.

<< fix / insert diagram >>

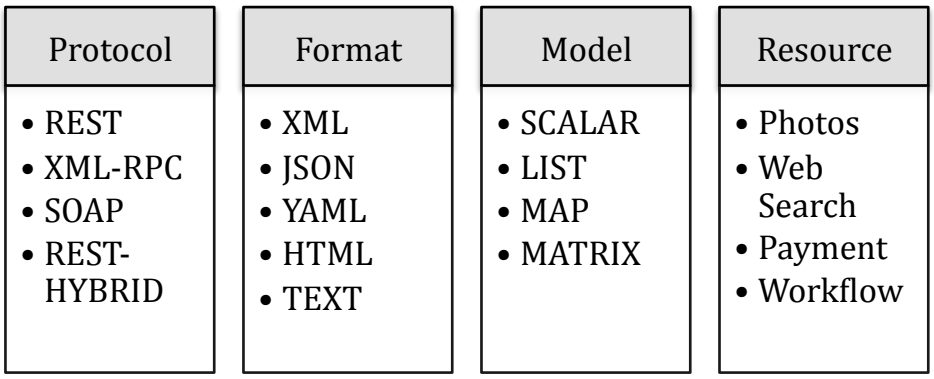

Figure 4.2: Module Overview

4.3 Statelessness

In WUT each request is handled in a stateless manner. This means that each request contains all the information necessary to execute the request. This allows requests to be handled by any application server regardless of where previous requests were handled. It does not mean that resources cannot save any data for subsequent requests, only that any data stored must be made accessible by all application servers. In other words, all state that needs to persist between requests is stored outside of the application server. The stateless nature of the application server

4.4 Protocols

The protocol is the interface between the developer and the resources made available through WUT. Protocols are written completely independently and depend on the `ProcessingPipeline` to process the request appropriately. Protocols that need to know what resources are available ahead of time use the `ResourceManager` to

discover what resources are available for consumption. In order to accomplish this, the protocol is responsible for the creation of a `RequestContext` which contains all the information necessary to process the request including the resource requested, parameters specified, requested format, and information about the user making the request.

WUT currently supports four different protocols: REST, XML-RPC, REST-RPC-HYBRID, and SOAP. All four protocols we chose to initial develop are web-service based. This means they all work over HTTP, which allowed us to easily load balance all of the protocols in a similar fashion. Even though only web-service based protocols were chosen for initial development, WUT protocols are not limited to working over the HTTP protocol. Support for other RPC based protocols like Java's RMI could easily be integrated into WUT.

Due to the fact HTTP works on top of TCP/IP, the four protocols are all based on top of TCP/IP as well. A diagram of the protocol stack can be seen below.

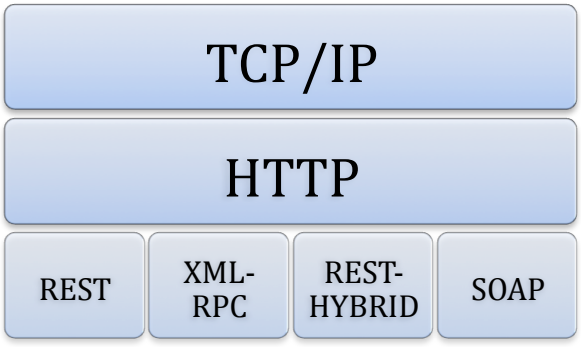

Figure 4.3: Protocol Stack (reverse stack direction)

All of the protocols used with WUT contain a payload with formatted model objects. A diagram showing how the data gets wrapped, converted, or formatted can be viewed below.

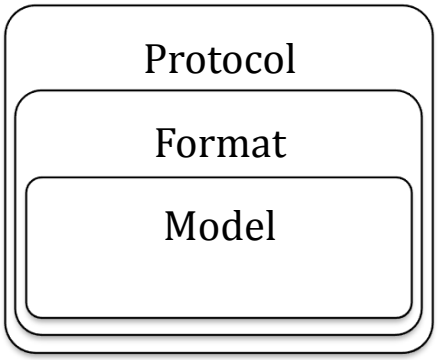

Figure 4.4: Data Transformation Process

Although the protocols communicate the same data, each will do so in a slightly different way. For example, a REST request for a random number may look like:

```
GET /resource/xml/random/number HTTP/1.1
Host: www.webutilitytoolkit.com
```

Figure 4.5: REST Request

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8

<?xml version="1.0"?>
<results>
  <integer>12345</integer>
</results>
```

Figure 4.6: REST Response

Note how all the information for the request is specified in the HTTP headers. This is traditional for REST. Also note how the body of the response has no particular standard. It simple conforms to the format requested (xml) and the model used by the requested resource (a simple integer scalar).

And the same request for a random number in XML-RPC might look like:

```
GET /xmlrpc HTTP/1.1
Host: www.webutilitytoolkit.com

<?xml version="1.0"?>
<methodCall>
  <methodName>random.number</methodName>
</methodCall>
```

Figure 4.7: XML-RPC Request

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value>
            <string>
                  <integer>12345</integer>
            </string>
        </value>
    </param>
  </params>
</methodResponse>
```

Figure 4.8: XML-RPC Response

<< FIX RESPONSE >>

Note how the XML-RPC request specifies the resource in the body of the request and wraps the response with additional xml syntax.

4.5 Formats

Format refers to the presentation of the data. The same data can be presented in many ways. For example, in XML a list of colors might be presented as follows:

```
<?xml version="1.0" ?>
<list>
   <string>blue</string>
   <string>red</string>
   <string>green</string>
</list>
```

Example 4.1 XML Format

That same list in JSON could be represented as:

```
["blue","red","green"]
```

Example 4.2 JSON Format

And in CSV (Comma Separated Values) it might just be:

```
blue, red, green
```

Example 4.3 CSV Format

Regardless of the format, it is important that data can be both encoded and decoded in the format. Data is encoded from model form and given to a protocol to respond with.

4.6 Resources

Resources are the logical grouping of data and operations that access and modify that data. Resources are the only component within WUT that maintain state between requests and state must still be stored external to the application server. Resources can range from generic data sources like hash or relational to domain specific data sources like web search results or photos. The appropriate resource to handle a request is chosen by the `ResourceManager`. Resources are given a `RequestContext` object that contains all the information a resource needs to complete a specific request including any arguments specified and the user making the request.

Each resource completes requests in a completely different way, unique the resource being requested. The following sections breakdown each resource individually and discuss their design including any services or libraries they use to handle the request. Since WUT resources must store between-request state externally, it is common for WUT resources to depend on other services for fulfilling requests.

4.6.1 Web Search

The web search resource works by making a request to the Yahoo Web Search API [38]. Results are returned as list of maps where each map contains the title, description, and URL of a website.

4.6.2 Photos

The photos resource works by making requests to the Flickr API [39]. The Flickr API takes care of searching, storing, and retrieving photos.

4.6.3 Geocoding

The geocoding resource works by making requests to the Yahoo Geocoding Service [40]. Results from geocoding are list of maps where each map contains a latitude, longitude, address, city, state, and zip code. If information in unavailable is it simply left blank. For instance a search for the location "san luis obispo" will not return an address, so the address will simply be left blank in the response.

4.6.4 Payments

The payments resource works by making requests to the Paypal NVP API [41]. Requests return either success or failure.

4.6.5 Relational Storage

The relational storage resource works by taking advantage of SimpleJPA [42], a Java persistence API for Amazon's SimpleDB [43]. SimpleDB is a scalable relational database management system (RDBMS) made available through Amazon's Web Services API [44]. SimpleDB varies from traditional RDBMSs in several important ways:

- No user specified meta-data
- Multiple attribute values
- No data integrity
- No transactions
- Limited functions

Not having any user specified meta-data to describe the objects it stores means that SimpleDB has no pre-defined types. In addition to no pre-defined types, SimpleDB allows users to store multiple attribute values for a single attribute, unlike traditional RDBMSs, which only allow a one-to-one mapping of attributes to values. Since SimpleDB doesn't store any meta-data about your objects, it likewise doesn't provide any integrity for data you store. It also lacks support for transactions and their associated atomicity, consistency, isolation, and durability [45]. This is not to say that SimpleDB statements are not atomic and durable though, as they are atomic and arguably more durable than that of traditional RDBMSs[2]. The last differentiator between SimpleDB and a traditional RDBMS is that of SimpleDB's limited operator support. SimpleDB supports

[2] SimpleDB replicates data between multiple computers (and their respective hard drives) insuring that if one were to fail the data would still be available.

the following operators: equals, not equals, less than, greater than, less than or equal to, greater than or equal to, and, or, is null, is not null, starts with, and order by [43].

WUT simplifies this model further by taking on the limitations of both SimpleDB and traditional RDBMSs. Limiting relational use in this way allows WUT to replace SimpleDB with another relational management system without breaking WUT's relational data interface. WUT provides a one-to-one mapping between attributes and values and supports the same limited operation set as SimpleDB.

4.6.6 Hash Storage

The hash storage resource works by utilizing an open source distributed hash table called Ehcache [46]. Although Ehcache is intended for the temporary caching of java objects, it supports both an eternal and disk storage mode. Eternal is a mode in Ehcache that allows your data to persist eternally and disk storage is a mode that tells Ehcache to save its to disk rather than keeping them all in memory. WUT runs Ehcache within the Apache Tomcat [47] servlet container connects to it via Ehcache's own REST protocol.

4.6.7 Email

The email resource works by connecting to any SMTP compatible email server to send email. Email requests require an email address to send the mail to, a subject for the email, and a body which represents the email's contents. The email resource depends on the user having several settings set including email.host, email.port, email.sercure, email.username, and email.password. The email.secure setting specifies whether or not the email resource should use SSL to connect to the corresponding email server host.

4.6.8 Scraping

The scraping resource simply makes an HTTP request for the specified website, uses the Tidy library to turn the HTML into XHTML (clean up the HTML, fix broken tags, mismatched tags, etc), and then, if specified, applies an XPATH selector to the results and returns them.

4.7 Creating a New Resource

Creating a new resource is a task that can only be completed by WUT administrators. Creating resources is intended to be a relatively easy task as it is expected that new resources will regularly be added to WUT.

In order to write a create a new WUT resource, WUT developers need to first write a resource class and annotate it with the `Resource` annotation. The `Resource` annotation specifies the resource's name, group, and description. An example of an annotated resource class can be seen below.

```
@Resource(name="new-resource", group="new-resource-group",
desc="describes what this new resource does")
public class NewResource {

}
```

Example 4.4 Resource Skeleton

This resource class acts as a shell for operations that related to the resource. To add operations to the resource developers create operation objects, which they annotate with an `Operation` annotation. A complete resource that includes an operation to read random numbers can be viewed below.

```
@Resource(name="number", group="random", desc="generates random
numbers")
```

```
public class RandomNumberResource {
   @Operation(name="read", type=OPERATION_TYPE.READ)
   private static class ReadOperation extends Operation {
      @Override
      public Data perform(RequestInfo ri) {
         int rand = new Random(System.currentTimeMillis()).nextInt();
         return new IntegerScalar(rand);
      }
   }
}
```

Example 4.5 Resource Implementation

After developing the resource it needs to be added to the `ResourceManager`. Simply add the resource's class (`RandomNumberResource.class`) to the `ResourceManager`'s static `resourceClasses` array and it will automatically be processed and added to the available resources during WUT's initialization routine.

4.8 Load Balancing

The load balancer is the component responsible for distributing application load across all of the individual application servers. The load balancer distributes load at the request level. It uses a simple round robin approach to decide which application server should handle a particular request. In addition to distributing the load, the load balancer also provides HTTP service testing. This means that if the load balancer is unable to establish a HTTP connection it will try the next server in its pool until its able to successfully make a connection. This helps to reduce service failure and provides N-1 fault tolerance.

When load becomes too much for any particular load balancer, WUT uses a technique referred to as DNS balancing to distribute load between multiple load balancers. DNS balancing involves rotating the DNS entries used for a domain. Since most clients start by requesting the top DNS entry this technique causes client requests to

be distributed across all of the DNS entries. It should be noted that this technique is susceptible to problems caused by caching of DNS entries. WUT has not addressed these problems at this time.

4.9 Summary

WUT provides ten essential resources commonly needed in web development and does so in a way that makes these resources easy to consume. WUT was designed with a lot more than these ten resources in mind though. WUT was built flexible and extensible enough to allow WUT developers to easily add new resources as they see fit. We hope that developers not only find use in the current resources provided by WUT but also in the resources yet to come.

5 KILLERFLICKER.COM

KillerFlicker.com is a website that was built as example of how to use Web Utility Kit (WUT). It allows users to upload and browse images as well as search public image repositories. It employs many WUT resources to complete this task including: photo, hash, and relational resources. It was built using the flash based client-side technology Flex and uses XML to communicate with WUT over the REST protocol. This combination was chosen for its brevity and descriptiveness; although the application could easily have been built using another client-side language, protocol, and format. This chapter walks through the process of building this application, concentrating on the elements of the application that consume WUT resources. A complete view of the final application can be found in Appendix B.

5.1 Project Overview

Before discussing the details of this project, we would like to introduce the functionality we'll be developing. This application will be performing the following tasks:

- Searching of Public Photos
- Uploading User Photos
- Viewing a Personal Library of Uploaded Photos
- Viewing of Search History

Each task will be given its own tab and corresponding component. All of the components will rely on an ActionScript class called `WutResource` for communication with WUT. The following is a picture of the outcome of this project.

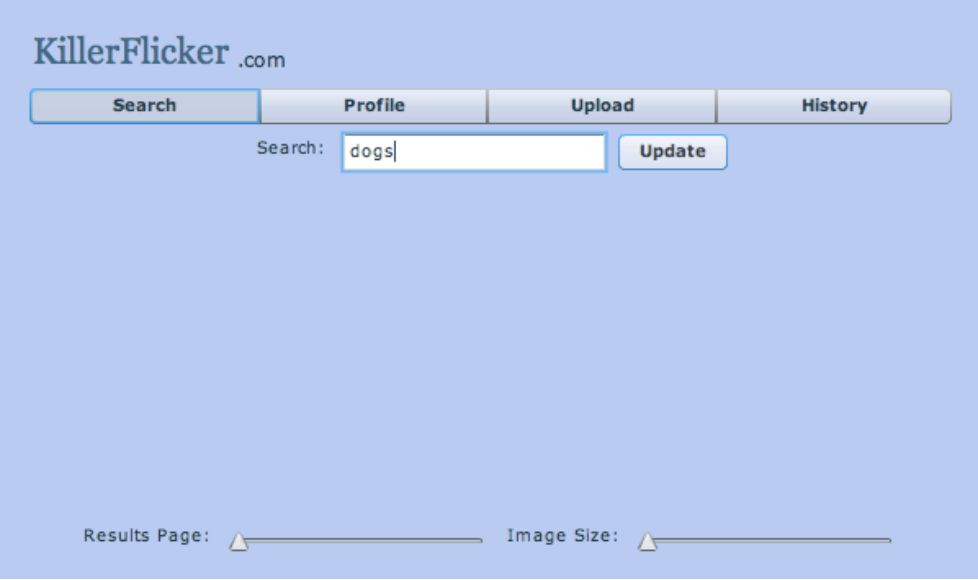

Figure 5.1 KillerFlicker.com Screen Shot

5.2 Application Frame

The application frame provides navigation between the four major application features: search, library, upload, and history. The following MXML file specifies the interaction and layout of those four components.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="vertical" xmlns:local="*">

   <!-- KillerFlicker.com Logo -->
  <local:Login />

   <!-- Navigation Bar -->
```

```
    <mx:ToggleButtonBar width="100%" dataProvider="viewstack" />

    <!-- Navigation Panels -->
    <mx:ViewStack id="viewstack" width="100%" height="100%">
        <local:Search />
        <local:Profile />
        <local:Upload />
        <local:History />
    </mx:ViewStack>

</mx:Application>
```

Example 5.1 KillerFlicker.mxml

As can be seen above, each application features is broken into its own Flex component. The following sections will each outline a different Flex component.

5.3 User Management

The login component takes care of user management. It contains two important methods: one that checks to see if the given user exists and another to create a user if the user needs to sign up for an account.

```
private function checkUser(username, password) : void {
    wut.loadResouce("GET", "hash", "data", "key=usernames." + username,
    function(data) {
        if (data.result.value == password) {
            hide();
        } else {
            error();
        }
    });
}
```

Example 5.2 CheckUser Function

The `checkUser` function makes a hash request to retrieve the password for a specified username. If result contains a matching password then it simply hides the login component and allows the user to interact with the rest of the application. If the result

does not contain a matching password then it displays an error. If the user doesn't have an account he or she can create one. Code for creating a new user can be seen below.

```
private function createUser(username, password) : void {
   wut.loadResouce("POST", "hash", "data", "id=usernames." + username +
"&value=" + password,
   function(data) {
        hide();
    });
}
```

Example 5.3 CreateUser Function

Upon successfully saving a new username and password to the hash resource the `createUser` function hides the login component and allows the user to interact with the rest of the application.

5.4 Photo Searching

The photo search component searches for public photos that have been tagged with a given set of keywords. It relies on two different WUT resources: photos and relational. The photos resource is used to search for public photos while the relational resource is used to save searches in a virtual table. Results of searching for photos can be viewed in a grid below the search input field and a history of previous searches can be viewed on the history tab.

To created this component two `WutResources` are used:

```
var relational : WutResource = new WutResource("relational ", "vrecords");
var photos : WutResource = new WutResource("photos", "search");
```

Figure 5.2 Photo Search Resource Declarations

Once the user hits the search button, the search input is validated and the update method is called. The update method uses the two WutResources declared previously to save the search terms and populate a grid with the search results.

```
private function update() : void {
  relational.setParams("data={terms:" + tags.text + "}");
  relational.create();
  photos.setParams("tags="+tags.text+"&page="+page.value+"&perpage=6");
  photos.read(function(data:XML) {
    photoRepeater.dataProvider = data.results.map;
  });
}
```

Figure 5.3 Photo Search Update Method

5.5 Library

The library component allows users to see photos they've previously uploaded. It only uses one WUT resource: photos. The following code is used to create this resource:

```
var photos : WutResource = new WutResource("photos", "");
```

Figure 5.4 Library Photos Resource

Immediately when the component is loaded, it calls an init method that requests the first page of photos.

```
public function init() : void {
  photos.read(function(data : XML) {
    photosRepeater.dataSource = data.results.map;
  });
}
```

Figure 5.5 Library Init Method

5.6 Photo Uploading

The photo uploading component is responsible for sending photos to WUT. For this it creates a single `WutResource` representing the photos resource. It will primarily use this resource to get a URL to send a multi-part HTTP message to.

```
var photos : WutResource = new WutResource("photos", "library");
```

Figure 5.6 Upload Resource

The user initiates an upload by clicking the "upload" button. The upload in turn opens a browse dialog allowing the user to choose a file. Once a file is chosen the `upload` method is called by the `fileUploader`, which in turn uploads the selected photo.

```
private function browse() : void {
  fileUploader.browse();
}

private function upload(evt:Event):void {
  try {
    fileUploader.upload(new URLRequest(photos.getUrl()));
  } catch (e:Error) {
    message.text = "error uploading: invalid file chosen";
  }
}
```

Figure 5.7 Photo Upload Browse and Upload Methods

5.7 WutResource

The `WutResource` actionscript class is the heart of the application. It performs all the communication with WUT. In this class we'll examine two crucial methods: `loadResource` and `loadUrl`. The method `loadResource` is responsible for building the appropriate URL corresponding to the method, resource, operation, and parameters provided.

```
function loadResouce(method:String, resource:String, operation:String,
params:String, callback:Function) {
      var url:String =  "http://" + host + ":" + port +
                        "/resource/xml/" + resource + "/" + operation +
                        "/" + params;
      loadUrl(method, url, callback);
}
```

Figure 5.8: WutResouce loadResource Method

As you can above, due to the simplicity of the REST protocol resources can be easily indentified using a single URL (and corresponding method).

The second method that we'll look at in this class is `loadUrl` method. This method makes an HTTP request for a given URL and calls the given callback function upon success. The callback function is given the results of the HTTP request as an argument.

```
function loadUrl(method:String, url:String, callb
   trace('requesting url ' + url);
        var request:URLRequest = new URLRequest();
        request.url = url;

        // For Non-GET requests use POST because DELETE is non-
idempotent
        if (method == "GET") {
           request.method = URLRequestMethod.GET;
        } else {
           request.method = URLRequestMethod.POST;
           // some data is required for POST requests to work correctly
           request.data = new URLVariables();
           request.data.foo = "bar";
        }

        // Specify X-HTTP-Method-Override header
        request.requestHeaders = [
            new URLRequestHeader( 'X-HTTP-Method-Override', method)
        ];

        var loader:URLLoader = new URLLoader();
        loader.addEventListener(Event.COMPLETE, callback);

        try {
           loader.load(request);
        } catch (error:Error) {
           trace("unable to load requested url.");
        }
```

```
}
```

Figure 5.9: WutResource loadUrl method

There are four important parts to this function: creation of the HTTP request, setting of the HTTP method, setting of the X-HTTP-Method-Override headers, and the loading of the URL. The Flex class URLRequest represents an HTTP request. This class is important because HTTP is the protocol upon which REST is built. REST also depends on the appropriate use of HTTP methods, but unfortunately modern web browsers (and therefore Flex) only support the HTTP methods GET and POST. Fortunately, WUT supports the use of the X-HTTP-Method-Override header, which allows the user to override the HTTP method used to make the request with one specified by the X-HTTP-Method-Override header value.

5.8 Summary

In this chapter we demonstrated how WUT can be used to create powerful web applications with minimal effort. We demonstrated use of several of WUT's key features and included a variety of resources in the process. Once finished, we had a basic photo sharing application capable of handling a large number of concurrent users, and did so in less than 500 lines of code.

6 ANALYSIS

In analyzing Web Utility Kit (WUT) we evaluated three different criteria: performance, scalability, and usability. To evaluate the performance of WUT we compare it against that of several of the services it consumes. To evaluate the scalability of WUT we evaluate the number of requests WUT can serve using a varying number of application servers. To evaluate the usability of WUT we conducted an informal focus group to gather feedback from initial users. For both performance and scalability testing we limited our testing to the REST protocol and the XML format. The nature of other protocols and formats may introduce fixed marginal variation, but since we're not interested in measuring the overhead of specific protocols or formats, we've ignored those differences in our analysis.

6.1 Performance Overhead

To evaluate the performance of WUT we compare it against the performance of comparable services. Since WUT consumes many other services in order to provide its resources (as can be seen in Chapter 4 DESIGN), those services provide an excellent way of evaluating the performance overhead incurred by WUT. The services used to compare against WUT include:

- Flickr API
- Google Web Search Service
- Gmail's SMTP Service

- Yahoo's Geocoding API

The graph below shows the average time to service a particular request for both WUT and its comparable service. For each comparison, the average time was taken over two hundred requests. Requests were made via the RESTlet client library [48] and performance measurements were taken using the Java™ Execution Time Measurement Library [49].

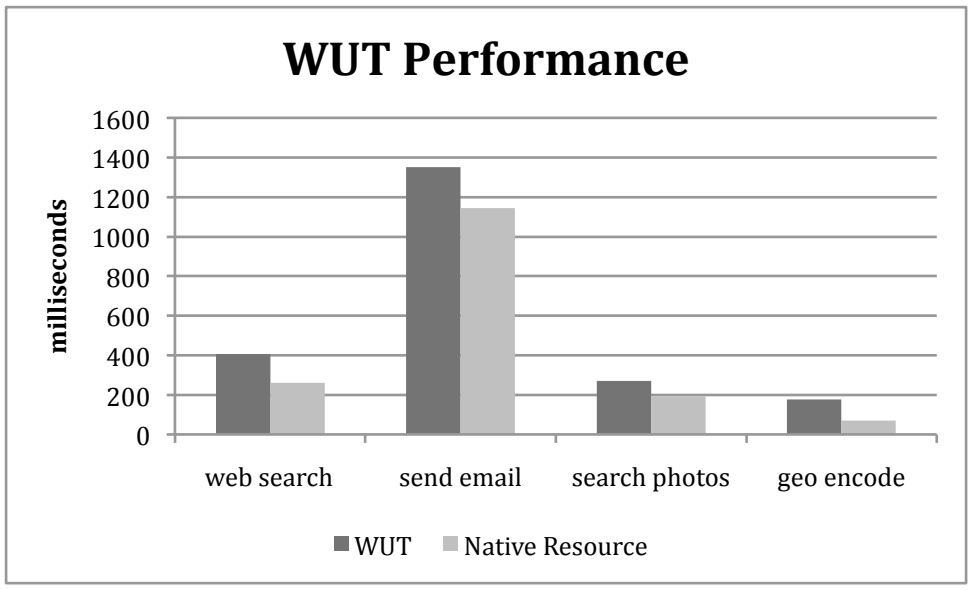

Figure 6.1 WUT Performance Comparison

As you can see by the graph, the time WUT takes to compete these requests is only marginally higher than that of the comparable service. On average WUT incurs an overhead of 134.7 ms over comparable services. This represents an 8% increase in overhead.

6.2 Scalability

To measure WUT's scalability we used the metric of how many requests per minute the service can handle. To measure this we used the tool Apache Bench [50]. Since Apache Bench is only capable of making requests to a single address at a time and no single address would accurately represent WUT we averaged the results of many Apache Bench trials to give us a more accurate picture of how many requests per minute WUT can handle. An example of the syntax used for a single trial can be seen below.

```
ab -t -x <resource url>
```

The process of aggregating individual results is then repeated with two, three, and four servers; each time measuring the number of requests per minute that WUT can serve. Results can be seen below.

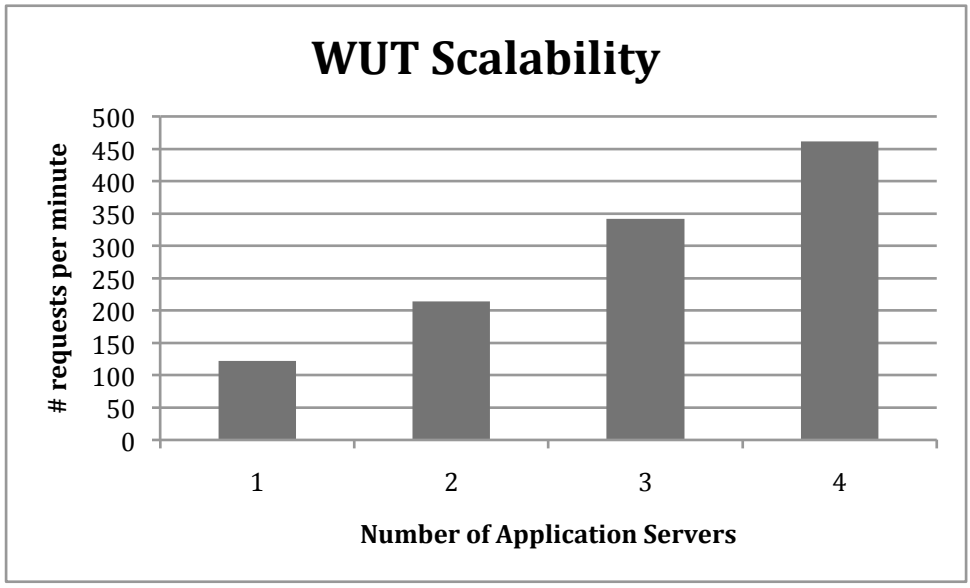

Figure 6.2: WUT Scalability

As can be seen from the graph, WUT accomplishes nearly linear scaling. This important property allows WUT to add resources to service more requests without having to worry about diminishing returns. It should be noted, however, that the capacity of the load balancer was not reached during this testing. In cases were this limit is encountered, DNS load balancing is used to balance between the load balancers.

6.3 Usability

Usability is an inherently difficult thing to measure. Rather than trying to contrive complicated methods or metrics to measure usability, we will instead simply provide feedback from initial users of the service. It is our opinion that the only way to truly measure the usability of a product or service is to listen to the people using it.

We let five different users use WUT for a two-week period and then hosted a focus group where we gathered the initial users together to discussed their experience. The five initial users came from a variety of computing disciplines including a compiler designer, video ingestion engineer, web application developers, and university student studying computer science. In the focus group we touched on topics ranging from performance to security, but feedback focused two major areas: ease of consumption and requested features.

Everyone in our initial user pool used either the REST protocol or the REST-RPC-HYBRID protocol for interfacing with WUT, despite the fact they were all encouraged to try all of the protocols available. People had a strong preference towards these protocols due to their "simplicity" [51] and "browser compatibility" [52]. Since every user in our focus group had a browser handy, they were able to start testing WUT using their browser right away. This lead to users being amazingly impressed by how

easy it was the consume WUT resources and how much they could accomplish in "amazingly short amounts of time" [53]. While users seemed to standardize on protocol, they all preferred a wide variety of formats. The format they used seem to depend mostly on the language in which the developed in. JavaScript users preferred JSON, while ActionScript and Java programmers tended to prefer XML. The single C programmer (our compiler designer) preferred CSV. People were very pleased with the selection of formats offered and nobody could come up with another format they'd like to it. Overall people were most impressed by WUT's ease of consumption. This was the feature people said made them "most likely to use the platform" [52].

Focus group members had a wide variety of feedback in respect to features they would like to see. The more popular suggestions included "more advanced SQL like functions" [51] for the relational resource, the ability to read email in addition to sending it, "word lookup features" [53] like that of an online dictionary service, and the ability to perform non-request based tasks. That last feature, non-request based tasks, was universally requested by all of our initial users. The types of tasks they'd like to perform are typically reoccurring batch-processing jobs. An example would bank website that ran a once a month task to e-mailed all of its users a statement of their bank usage for the month. Since currently WUT can only respond to requests, WUT has no way of performing that kind of processing.

6.4 Summary

We are pleased by the initial performance, scalability, and response received by WUT. We feel as though an 8% increase in overhead represents a marginal increase and

is well worth the benefits received by using the service. We feel as though WUT's ability to scale linearly represents a great ability to grow as usage demand. In addition, the overwhelmingly positive support from our initial user base tells use that WUT has achieved an acceptable level of usability for its initial release.

7 RELATED WORK

In this section I'll discuss several projects with a related intent to the work we've developed: a scalable service that simplifies the process of developing dynamic websites. The three services we'll cover are Pyshards, OpenDHT, and Google AppEngine. We have developed the following characteristics as important to the usefulness of the service or framework:

- How the service is consumed
- What languages it supports
- Whether or not its hosted service
- The level of transparency of it's distributed nature
- Any limitations which would hinder its usefulness

Each of these characteristics will be mentioned for services and frameworks below including how they compare to that of WUT.

7.1 Pyshards

Pyshards is a MySQL based Python library for the horizontal partitioning (sharding) of relational data [35]. It's capable of automatically distributing SQL data into shards and then querying data from individual shards or collectively aggregating data from multiple shards. It provides a subset of the SQL-92 standard to access data and eventually hopes to be able to support the entire standard. Pyshards allows for scaling up

to the current number of nodes squared before requiring rebalancing and it capable of rebalancing automatically once required [35].

Pyshards is limited to use with Python, only provides relational storage, and is limited in interface to that of SQL. WUT on the other hand supports a variety of protocols, formats, and resources making it easier to consume from a variety of languages and much more flexible in terms of what can be built using the service. WUT also provides a syntax that alleviates the end user from the need to use SQL altogether thereby simplifying the development process for simple applications. To further add to the benefit of using WUT, due to its hosted nature, it also eliminates the need to install, maintain, and support a complex database management system and corresponding servers.

7.2 OpenDHT

OpenDHT is a decentralized distributed system that provides a lookup service similar to that of a traditional hash table. It works by partitioning the keyspace into different ownership keys among the participating nodes, which are connected through an overlay network [37]. The system focuses on decentralization, scalability, and fault-tolerance. The system is made available as a service to the public accessible via SunRPC and XML-RPC, which makes it suitable for consumption by nearly any language. Values stored however are only stored for a limited amount of time, referred to as the value's time-to-live (TTL), after which point the value will be deleted. It is up to the client application to restore any data it feels necessary to keep. There is also a limit placed on the amount of data that any particular IP address can store. In addition to the per IP address storage limit, there is also a limit for amount of space any particular IP address

can use on one disk. This per disk limits abuse of any particular disk and rewards clients that choose keys that are better distributed [37].

As WUT is not intended to be a free public service, it doesn't need place many of the usage limits required by OpenDHT. WUT makes no limit on the amount of data one can store or the length of time that data is for. In addition, WUT supports a wider range of protocols and formats, which increases usability by allowing developers to use the protocol and format with which they are most familiar. WUT also supports a larger variety of storage methods, allowing the user to store their data in the method most appropriate to them. For example, if they can choose to store their data in the relational data resource and in turn have the ability to select, project, sort, merge, and filter their data.

7.3 Google App Engine

App Engine is a framework that lets developers take advantage of Google's massively scalable infrastructure to develop and deploy web applications [36]. The goal of the project is to make it easy to get started with a new web application, and then make it easy to scale when that application reaches the point where it's receiving significant traffic and has millions of users [36]. The project has dynamic web serving with automatic scaling and load balancing (including caching). It provides a number of services including: hash based storage, email, url fetching, image manipulation, user management, and access to Google's Data Services (which includes content stored on Google Spreadsheets, Google Contacts, Google Documents, Piscaweb, and Youtube). They achieve scalability in many of these services by leveraging technology they

developed to support their own web applications including BigTable [54], GFS [55], and their robust email system that supports Gmail [56].

Since applications written for App Engine get run within Google's environment, it can be challenging to develop, debug, and test your application. To help solve that problem they provide a local toolset that emulates the functionality of their live industrial scale services [36]. This toolset allows developers to work locally and later commit their work to the live service.

Currently, there are several limitations associated with this framework. Google limits you 500 megabytes of total storage, 200 million CPU cycles per day, and 10 gigabytes of bandwidth per day. It is expected that in the future users will be able to pay for any usage beyond this criteria. Google estimates that most web applications will be able to serve around 5 million pageviews per month off the base allotment alone [36].

In addition to the usage restraints, there are also several implicit limitations of their framework. Since it's heavily tied to their infrastructure, it only runs on their servers. Developing an application for App Engine locks you into using their hosting service and infrastructure for the life of your application. You are limited to only being able to develop in Python. Your project must be web based. You have no control over the services offered.

App Engine is perhaps the closest competitor to that of WUT. Both App Engine and WUT offer a variety of services commonly used by web developers and do so in a scalable manner. Only WUT, however, allows the developer to consume its services using the protocol and format of their choice, allowing for a language independent service. App Engine provides a Python specific API to use their service, which hinders

development for users not already familiar with Python. In turn, WUT maintains a clean separation between server and client and is careful to only provide server functionality while leaving the client technology choice up to the developer. App Engine on the other hand muddles the two together.

In addition to their architectures, WUT and App Engine also differ on the services they offer. App Engine offers two services for which WUT currently has no equivalent: calendar and video, while WUT offers two services for which App Engine has no equivalent: payments and geocoding.

7.4 Comparison

The following tables offer a side-by-side comparison of the features provided by each of the services and frameworks described above.

7.4.1 Protocols

	Pysards	OpenDHT	AppEngine	WUT
REST	NO	NO	NO	YES
XML-RPC	NO	YES	NO	YES
SOAP	NO	NO	NO	YES
RMI	NO	NO	NO	YES
SunRPC	NO	YES	NO	NO
Native Python	YES	NO	YES	NO

7.4.2 Formats

	Pysards	OpenDHT	AppEngine	WUT
XML	NO	YES	NO	YES
JSON	NO	NO	NO	YES
YAML	NO	NO	NO	YES
Plain Text	NO	NO	NO	YES
HTML	NO	NO	NO	YES
Native Python	YES	NO	YES	NO

7.4.3 Resources

	Pysards	OpenDHT	App Engine	WUT
Photo	NO	NO	YES	YES
Photo Editing	NO	NO	YES	YES
Web Search	NO	NO	YES	YES
Relational Data	YES	NO	YES	YES
Hashed Data	NO	YES	YES	YES
Email	NO	NO	YES	YES
URL Fetching	NO	NO	YES	YES
Video Search	NO	NO	YES	NO
Payments	NO	NO	NO	YES
Geocoding	NO	NO	NO	YES
Calendar	NO	NO	YES	NO

7.5 Summary

WUT's provides a competitive resource offering that includes resources similar to most of the services described in this chapter. WUT also contains the ability to consume other services through the creation of new resources, of which any of these comparable services could be included. The closest competitor to WUT, Google's AppEngine, is only available to python developers and lacks support for many of the protocols and formats made available by WUT. Due to this fact, we feel WUT has a strong position in the hosted platform market and will provide a valuable asset to web application developers.

8 CONCLUSIONS

This chapter discusses how Web Utility Kit (WUT) has contributed towards making the development of web applications easier, more efficient, and more scalable. It reviews WUT's contributions, usability, performance, scalability, and initial user reactions.

8.1 Contributions

WUT is the first hosted platform that unites resources to make them completely consumable by client-side technologies. In the process of achieving that goal it has also become the first framework for making resources accessible from all of the major web-service protocols. It provides a unique set of resources that are common to web development, helping to increase productivity and decrease deployment time of modern web applications.

8.2 Ease of Use

WUT has been shown to consolidate resources that might otherwise take developers significant time to learn and integrate. It has also been shown to work with many of today's most common protocols and formats. Initial users have given WUT upstanding reviews in terms of its ease of use.

8.3 Performance and Scalability

Exhaustive testing has shown WUT's overhead to average less than 200 ms per request. This low overhead includes receiving the request, unformatting the any request

parameters, modeling the results in a common model, formatting the results, and returning the results to the application. Not only has WUT been shown to have a relatively low overhead, but it has also been shown to be able to scale nearly linearly. Testing revealed WUT capable of handling almost 500 requests per minute with less than five application servers (and corresponding resource servers).

8.4 Usefulness

The initial set of users has indicated WUT to be extremely useful for a variety of applications. User comments ranged from "would speed development time up by a thousand times" [51] to "the most useful web framework I've ever used" [53].

9 FUTURE WORK

Despite the achievement made by this work's contribution, there are still myriad features with which this project would benefit. This chapter discusses several areas that would benefit from future work included caching, enhanced resources, data validation, resource composability, usage monitoring, scheduled processing, and development tools.

9.1 Caching

There are multiple places in the WUT architecture where caching could be of benefit. The two places that seem to be the most obvious are at protocol stage and model stage.

At the protocol stage, caching of previous requests could result in directly returning formatted results, completely skipping the whole process of formatting, modeling, and fetching of data. This would however only apply to idempotent requests that share the same format.

Caching at the model stage allows for format independent caching, which saves from the need to fetch the data, but still requires formatting of the data. This allows for multiple requests with different formats to share the same cache, increasing the chance of cache hits, but decreasing the response time do to the need to reformat the data. It is likely that this caching would apply best to resources that are frequently shared amongst different users, since a particular user is likely to choose a single format for all their needs.

9.2 Enhanced Resources

There are an infinite number of resources that could be beneficial to developers. Resources that have been requested by initial users for inclusion in WUT include image manipulation, captcha generation, word definition and synonym retrieval, advertizing generation, and map retrieval. There are also a number of improvements that could be made to current resources. The relational resource would benefit from the ability to perform functions like summing, averaging, or concatenating result; grouping of results; and limiting the number of results. The email resource would benefit from the ability to read email as well as send email.

9.3 Data Validation

Currently each individual resource is responsible for validating its own arguments. A shared way to validate data would make developing new resources easier and also help to standardize the errors returned from missing or invalid arguments. In order to accomplish this shared data validation, resources would need a way to specify the data types they wish to receive and what type of data they will output.

9.4 Resource Composability

Resource composability would allow for resources to be used in conjunction to both simplify and speed up use of multiple resources. This idea is similar to how applications are piped in Unix. The output of one resource would be used as the input to another resource. This chaining could be as deep as the developer desired. This would save applications from having to make more than one request and also save the round time of transferring the data between server and client. An example use of resource

composability might include having all the data in a relational "tasks" table marked as pending be emailed to you.

9.5 Usage Monitoring

In order to ensure "fair" use or charge for a service like WUT, you must be able to monitor its usage. Currently WUT doesn't monitor usage at all. A new module that globally stored user and usage information could be added to the `ProcessingPipeline` and that would help to resolve this deficiency.

9.6 Scheduled Tasks

WUT would benefit from the ability to schedule a task to be completed at a later point in time. This would give applications a way to make run tasks aside from on a per-request basis. An example of a schedule task might be marking a relational entry as old 30 days after it's been entered.

9.7 Developer Tools

WUT would benefit from a set of tools that helped developers to accomplish common development tasks. These tools would be especially useful for interactions that happen only once or very infrequently. An example of this type of interaction might be the creation of a relational table or the management of WUT settings. The following is a list of developer tools that WUT might benefit from:

- Relational data editor
- Scheduled task editor
- Performance monitor
- Usage monitor

- Settings editor
- Error log viewer

10 Bibliography

[1] P. Clemente, *The State of the Net*, Mcgraw-Hill, 1998.

[2] M. Jazayeri, "Some Trends in Web Application Development," *2007 Future of Software Engineering*, IEEE Computer Society, 2007, pp. 199-213.

[3] R. MacManus, "The Evolution of Corporate Web Sites," *Digital Web Magazine*, Apr. 2004.

[4] T. Bourke, *Server Load Balancing*, O'Reilly Media, Inc., 2001.

[5] T. Schlossnagle, *Scalable Internet Architectures*, Sams, 2006.

[6] K. Swenson, "The Key to SOA Governance: Understanding the Essence of Business," 2008.

[7] D. Winer, "XML-RPC Specification," Jun. 1999.

[8] "SOAP Specification."

[9] R.T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, Irvine, 2000.

[10] L. Richardson and S. Ruby, *RESTful Web Services*, O'Reilly, 2007.

[11] "UDDI Version 3.0.2."

[12] R. Chinnici, J. Moreau, A. Ryman, and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," Jun. 2007.

[13] M. Hadley, "Web Application Description Language (WADL)," Nov. 2006.

[14] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, 2009, pp. 50-55.

[15] Krissi Danielson, "Distinguishing Cloud Computing from Utility Computing - SaaS Week," Mar. 2008.

[16] "Scalability - Wikipedia, the free encyclopedia," Mar. 2009.

[17] "Multitier architecture - Wikipedia, the free encyclopedia," Mar. 2009.

[18] "Client-server - Wikipedia, the free encyclopedia," Mar. 2009.

[19] Kyle Cordes, "YouTube Scalability Talk - Kyle Cordes," Jul. 2007.

[20] Randy Shoup, "InfoQ: Scalability Best Practices: Lessons from eBay," *InfoQ*, May. 2008.

[21] Jeff Lash, "Digital Web Magazine - Designing for Scalability," *Digital Web Magazine*, Jun. 2004.

[22] "Load balancing (computing) - Wikipedia, the free encyclopedia," Mar. 2009.

[23] "Cache - Wikipedia, the free encyclopedia," Dec. 2008.

[24] "Distributed database management system - Wikipedia, the free encyclopedia," Jan. 2009.

[25] "Partition (database) - Wikipedia, the free encyclopedia," Dec. 2008.

[26] "Shard (database architecture) - Wikipedia, the free encyclopedia," Mar. 2009.

[27] "Google Architecture | High Scalability."

[28] "Flickr Architecture | High Scalability."

[29] "Amazon Architecture | High Scalability."

[30] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin,

"Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, El Paso, Texas, United States: ACM, 1997, pp. 654-663.

[31] Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi, "Web Caching with Consistent Hashing."

[32] "Sun Servers."

[33] "Oracle Price List 2007."

[34] "Price Lists | Global Pricing and Licensing."

[35] "pyshards - Google Code."

[36] "Google App Engine - Google Code."

[37] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu, "OpenDHT: A Public DHT Service and Its Uses," *ACM SIGCOMM 2005*, 2005.

[38] "Yahoo! Search Web Services - YDN."

[39] "Flickr Services."

[40] "Yahoo! Maps Web Services - Geocoding API."

[41] "PayPal NVP API Overview - PayPal."

[42] "SimpleJPA."

[43] "Amazon SimpleDB."

[44] "Amazon Web Services."

[45] "ACID - Wikipedia, the free encyclopedia."

[46] "Ehcache Ehcache - ehcache."

[47] “Apache Tomcat - Apache Tomcat.”

[48] “Restlet - Lightweight REST framework.”

[49] “Java(TM) Execution Time Measurement Library.”

[50] “ab - Apache HTTP server benchmarking tool - Apache HTTP Server.”

[51] Josh Tang, “WUT Focus Group,” Mar. 2009.

[52] Rich Fuhler, “WUT Focus Group,” Mar. 2009.

[53] Alexander Page, “WUT Focus Group,” Mar. 2009.

[54] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, “Bigtable: A Distributed Storage System for Structured Data,” *ACM Trans. Comput. Syst.*, vol. 26, 2008, pp. 1-26.

[55] S. Ghemawat, H. Gobioff, and S. Leung, “The Google file system,” *SIGOPS Oper. Syst. Rev.*, vol. 37, 2003, pp. 29-43.

[56] K. Ivens, *The Gmail Book: Your Definitive Guide*, Prentice Hall PTR, 2005.

11 Appendix A

12 Appendix B: KillerFlicker.com Source Code

13 Appendix C: Oracle Software Pricing Calculation

Assuming you want a two-node RAC cluster consisting of 4 CPUs (ignoring multi-cores) per node plus the following options:

- Real Application Clusters ($20,000 per processor)
- Active Data Guard ($5,000 per Processor)
- Partitioning ($10,000 per Processor)
- Real Application Testing ($10,000 per Processor)
- Advanced Compression ($10,000 per Processor)
- Total Recall ($5,000 per Processor)
- Advanced Security ($10,000 per Processor)

Taking the options into account, the software cost per CPU would be US $110,000. For the 2 Node RAC, that would come to be (2 nodes) x (4 CPU) x $110,000 = $880,000.

The secondary site where your Active Data Guard is being replicated is missing. That would be double your cost to $1,760,000.

Likewise, adding support of 22 percent brings the figure to US$2,147,200 (list).

You would not pay this price, as discounting has not been applied. Using figures form Oracle's Store, we get a discounted figure of US$1,610,400.