UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

COS301 Mini Project

Software Requirements Specification

Group 5B

Keagan Thompson *u13023782*
Matthew Russell *u12047822*
Maret Stoffberg *u11071762*
Patience Mtsweni *u11116774*
John-Latham van der Walt *u13222580*
Abrie van Aardt *u13178840*
Johan van Rooyen *u11205131*
Hlavutelo Maluleke *u12318109*

Github Repository
https://github.com/keagsthom/COS301_Group5B_Phase2

1 History

Date	Version	Description
04-03-2015	Version 0.1	Document Template created - Keagan
04-03-2015	Version 0.2	Added Name and Student Number - Keagan
04-03-2015	Version 0.3	Added Name and student Number - Matthew
04-03-2015	Version 0.4	Added Name and Student Number - Maret
04-03-2015	Version 1.0	Added to my section - Maret
05-03-2015	Version 1.1	Added Name and Student Number - Patience
05-03-2015	Version 1.2	Deleted BitBucket folder and recompiled PDF - Keagan
06-03-2015	Version 2.0	Updated file - Latham
06-03-2015	Version 2.1	Updated - Latham
06-03-2015	Version 2.2	Updated - Latham
07-03-2015	Version 3.0	Added section 3.1.1 - Human Access Channels - Matthew
08-03-2015	Version 3.1	Update Quality requirements - Maret
08-03-2015	Version 3.2	Update sections - Maret
09-03-2015	Version 3.3	Added to section 3.3.(Integration Requirements) - Patience
10-03-2015	Version 3.4	Fixed a tag that was stopping the compilation of the PDF - Keagan
10-03-2015	Version 3.5	Added my sections - Abrie
10-03-2015	Version 3.6	Added my final part - Abrie
10-03-2015	Version 3.7	Added rest of my sections - Matthew
10-03-2015	Version 3.8	Adding Architecture Tactics - Keagan
10-03-2015	Version 4.0	Cleaned up the PDF a bit and checked Quality Requirements - Keagan
10-03-2015	Version 4.1	Added names to title page - Johan
11-03-2015	Version 4.2	Removed/Merged duplicate sections - Matthew
11-03-2015	Version 4.3	Updated history section - Keagan

Contents

2 Introduction

This is Phase 2 of the buzz-system we are going to as specified by the client. This document goes in more depth on some specifications that are required.

2.1 Purpose

2.2 Background

3 Architecture Requirements

3.1 Architectural Scope

- **Concurrent processing environment:** Buzz will have to run on a multithreaded environment, we need multiple users to be connected and exchange messages concurrently. This has to run on one or more concurrent managers.

  Buzz will run on a Cluster environment:
  Multiple computers will share a common computer server. The cluster that could possibly be used is IBM High Availability Cluster Multiprocessing. It runs on multiple computers and when one fails the other takes over.

  This provides the following benefits to Buzz system:

  - High performance: improve concurrent processing throughput.
  - Adaptability: the ability to integrate with platform-specific batch queue and load-balancing systems to maximize concurrent processing performance on a particular platform.
  - Fault Tolerance: when other nodes/threads fail, the system should continue running. It is easier to administer the entire system and concurrent managers.

- **Temporary or permanent storage environment:** Temporary storage environment: Concerning user data, all information will be cached into the browser. UserID, password, etc can all be stored in cookies.

Permanent storage environment: Database and use of flash drives, this is to store system images.

- **Scheduling:** After some intervals the system must be able to store its image to avoid data loss in case of power shutdown or other causes. This could be done, or stored in some non-volatile storage, could be in a hard drive. After system reboot all the data can be simply recovered whenever there occurs any problem.

- **Integration environment:** For a basic integration environment, client-server architecture would be deployed. System management server would provide single interface through which administrators can explore and manage metadata repositories. We can manage data with some studio (actual environment where data will be stored and integrated with other information) which provides a single point of control user and system data. With appropriate access and integration channels the studio and system management would communicate through multiple servers.

- **Persistence infrastructure:** When system shut-down, Buzz state must be maintain with persistence infrastructure. With the Database management systems (DBMS), we can achieve persistence and also services such as queries, auditing and access control. Persistence can also be achieve with system images: after some scheduling system image must be stored. State changes made to a system after its last image was saved are lost in the case of a system failure or shut-down. There are other persistence techniques that can be well integrated with Buzz, such as journals and dirty writes.

- **Providing a reporting infrastructure:** All global reports must be stored in a file system, created with JAVA programming language. This would be done with the created user interface (UI). Alternatively, depending on the integration channels and their protocols all reports that need to be generated can be store in the integrated database.

XML - this can be used to store temporary reports that need to be delivered in a certain time interval. This reduces access time and easy to maintain.

3.2 Access Channel Requirements

3.2.1 Human access channels

Human users should be able to access the system from all major devices and software. These include Windows, Linux, Mac OS, Windows Mobile, Android, and iOS devices. Human users will be able to access the system via thick desktop applications, as well as thin web clients.

However it is expected most access will be via thin web clients. The system may be accessed via all major browsers, namely Internet Explorer, Mozilla Firefox, Google Chrome, Opera and Safari. The system will comply with the basic W3C requirements to accomplish this goal.

The REST architecture is the primary candidate to implement communication between users and the system, with a backup choice of the SOAP architecture.

3.2.2 System access channels

In the scope of this project, no systems will require access to Buzz. However, when Buzz is integrated with Hamster, it should provide the necessary API (over HTTP) to interface with the Hamster Marking System.

3.2.3 Integration channels

The system must be able to integrate with the University's CS-LDAP (Light-Weight Directory Access Protocol) server. As well as the server which will store the uploaded media, such as images, videos, PDFs and the like. The system will therefor need to handle the LDAP, HTTP, SMTP (or other email protocols) and SMS protocols. The Buzz system will be integrated into the CS website such that when a user logs into the CS website, they will also be logged into Buzz System automatically.

3.3   Quality Requirements

- **Scalability:** The software system must be able to handle the registration and data of all the users registered on the LDAP server. If several users attempt to do the same action simultaneously, like commenting in a thread, the actions should appear to happen one at a time. The system must not have a maximum or minimum amount of users that may use the system at a time.

- **Performance requirement:** The system must have a fast response time. The system must be able to handle all the users and not slow down.

- **Maintainability:** The system must be easy to update and maintain.

- **Reliability and Availability:** The system must be usable on any modern web browser and operating system.

- **User Security:** All the user's information must be secure and only be visible or editable for users with the required permissions. All the data must be inaccessible for non-users.

- **General Security:** The integration interface must be secure, i.e. any information obtained from the LDAP server about any user should be used in the Buzz system only.

- **Monitorability and Auditability:** A log of all user activity must be available for the administration. The information that must be available for auditability are: users and their details, posts, deleted messages, and all actions performed.

- **Testability:** An interface must be available to certain staff of the system to enable extreme case testing of the system's functionality, and the actual user interface must be thoroughly tested to decrease the likelihood of a user finding a fault. In order for these two types of testing to be conducted effectively, the system must be made as controllable and observable as possible.

- **Usability:** The Buzz system must be easy to use. It can make use of UI metaphors like a having a "post-it" symbol for the post operation or an envelope for the messages link. The most common used operations must perform very quickly. The system must be easy to learn so new

users can adapt to it without help. The validation and error messages must be clear and understandable.

- **Integrability:** The system must be integrated with the LDAP server. All the data on the server like names, photos registered modules, etc. are used in the buzz system.

3.4 Architecture Constraints

The architecture constraints were indicated in the briefing document and the following is a list of technologies we are going to use in the project:

- Linux (Operating System): Linux is more stable than windows on servers. It is free and does not need added costs. Thus it makes a good OS to use. There is a lot of free software that runs on it, but there is also software that only runs on Windows. The client wishes to use Linux, an open source OS, does not have that great customer service, but the free software and the abundance thereof makes up for it. The end-product should also be able do run on small devices, so it must be lightweight in resource usage.

- Git (Version Control System): Git is good for group projects and information share. It can show the progress and the group's efforts in completing the project. It is not always easy to use, sometimes it can break and work need to be restored from a previous version, and not all people know how to use it which could lead to faults in the development of and or slowing down progress.

- JavaEE (Java Platform Enterprise Edition): For server programming Java EE would suffice, it is reliable and easier to use. Even though it is a bit slower and is resource intensive. It makes a good framework that incorporates JPA, JPQL and JSF. Thus development can only be done on capable machines and not things like tablets or phones.

- JPA (Java Persistence API): JPA uses JPQL so it would be better to use this, it operates against entity object rather than with database tables. Most of the developers have not yet practised with this technology which leads to inexperience with this. The development of ht UI will be somewhat straining and therefore a capable machines must also be used.

- JPQL (Java Persistence query language): Is used to make queries against entities stored in a relational database and additionally retrieve objects from the database on the server. It would be used to retrieve user data and etc. Most of the developers have not yet practised with this technology which leads to inexperience with this. Databases can be big, but the amount of data through queries won't be too high, thus if internet connection is available, small devices could use this indirectly through the product.

- JSF (Java Server Faces): Is a Java specification for building component-based UI for web applications and exposing them as server side Polyfills. This works with AJAX. Most of the developers have not yet practised with this technology which leads to inexperience with this. In the development phase this is resource intensive thus again capable machines will be needed. The end product should be able to be supported on small devices and Operating Systems.

- HTML (Hypertext Mark-up Language): This is going to be code that has to be generated for the website where the user can navigate through, the browser takes this code, interprets it and give a UI for the user if it is a web-page (on a website). This is supported on almost all devices which make use of the internet.

- AJAX (Asynchronous JavaScript and XML): It is easier to use, rather than having to make our own function for everything. This is client side processing so that we dont hassle the servers with little tests and leave only queries and logging validation to the server. Thus internet usage is made less, and most of the work can be done on the device/machine.

- CSS (Cascading Style-sheet): This is the coding platform we are going to use to make the website more attractive and to fill the emptiness of the website if need be. Otherwise JSF would take the responsibility of this.

4 Architectural patterns or styles

4.1 Layering Architectural Pattern

The layering pattern allows for separation of concerns between the different layers. Due to each layer having its own responsibilities the system becomes very flexible and layers can be re-used due to the de-coupling. For example the front-end of the system can be separate from the infrastructure and how services are handled.
Possible layers to implement:

- Human/Client Access Layer

- System Access Layer

- Services Layer

- Infrastructure Layer

The layers can either communicate with components within the same layer, or with components in the next layer down. This provides the connection between the layers, and manages the communication in a structured fashion. The channel in which this communication can be achieved is via message passing between the layers.

4.2 MVC Architectural Pattern

Model-view-controller (MVC) is a software architectural pattern. When the MVC pattern is applied, an application is divided into three logical, interconnected parts. These parts are meant to isolate the presentation, business logic and data access aspects of the application. Adhering to this pattern results in implementations that are highly scalable and robust. Individual, M-V-C, components of the system can be physically disjoined and can account for less maintenance and housekeeping at each of their servers. This will aid in the separation of concerns: Back- and front-end modules can be neatly separated and this translates into a better, more parallel, development environment.

4.3 REST Architectural Pattern

The Representational State Transfer (REST) architectural pattern is an already highly used and tested pattern for scalable web services. The REST pattern functions by a coordinated set of constraints that govern how the Client and Server interact, and this specified set of constraints creates properties such as performance, scalability and modifiability. It is a stateless pattern, whereby there is no context stored on the server about the client. Since the user base of the Buzz system can be expected to grow to quite a large number, a RESTful approach would constitute less network hogging. This would also lead to a much simpler design since the server will only keep little session information about the client.

5 Architectural tactics or strategies

- **Functional Decomposition** By decoupling functionality, the system has functions that are smaller and a lot more specific, resulting in a great deal more scalability as you have more flexibility to scale them independently of one another at a lower resource cost. Additionally, this allows for greater availability as functions can now run to completion even if another function encounters errors in execution, assuming that error does not crash the system.

- **Virtualisation at All Levels** By creating multiple terminals out of the hardware, the system can adequately be shared between multiple users and this will no doubt extend the scalability of the system as more users will be able to efficiently access and use the system, without the entire system crashing if one user causes their part of the system to go down/stop execution.

- **Use Concurrency** When dealing with the use of resources, the access time to acquire them is a factor but blocked time is a more severe problem when multiple users are on a system at once. By allowing requests to be processed in parallel, blocked time is reduced which improves the system's performance and thus speeds up the user's work.

- **Employ Active Redundancy** Our system promises to be available to students whenever they need it, so availability is a big quality that we need to guarantee. Redundant components, which respond to events

in parallel, are all in the same state. When a request is made, only the fastest response is needed, even though the rest of the components produce the same response nevertheless. This is done so that should one of these components fail, virtually no downtime is experienced by the system because it will still receive a response from one of the components which are still up, so availability is not a problem.

6 Use of reference architectures and frameworks

There exist many reference architectures and/or frameworks which we can incorporate and use in our project. The API's already exist and can mainly be used as simple plugins.

Java-EE, which is a Layered Reference Architecture would be best suited for the current system, because it comprehensively includes a majority of our Quality Requirements that we want to achieve (like Scalability, Security etc).

Apache Tapestry (`http://tapestry.apache.org/`) is a "Component oriented framework for creating dynamic, robust, highly scalable web applications in Java." and can thus be used to address quite a few of the Quality Requirements. It also separates java from html code, making the implementation thereof that much easier and more effective to work with.

Other frameworks and API's follow:

- **Check for Plagiarism** Plagiarism-Checker (`https://github.com/architshukla/Plagiarism-Checker`) searches online by using Google Search API's to address queries. The URL you get from the query are checked for similarities with the information you are checking. Results are stored in an output text file. Checking user posts for plagiarism is of cardinal value, since it helps grow an air of responsible sharing in the users.

- **Highlighting Syntax** SyntaxHighlighter 3.0.83 (`http://alexgorbatchev.com/SyntaxHighlighter/`) is a self-contained syntax highlighter specifically for code. It is used in, among others, Apache, Mozilla, Wordpress and Freshbooks. It is developed in Javascript and works no matter what

kind of server you are running. Using this plugin can ease the process of sharing code in the discussion area and help see what is part of syntax and what is not.

- **Profanity Filter** A profanity filter is a must to maintain order and control in an online discussion area. Without it, users can post what they like, when they like without any sort of service or function prohibiting their possibly damaging posts. PurgoMalum (`http://www.purgomalum.com/`) is a free web service for filtering content. It can filter out curse words, other obscenities and general unwanted words as well. WebPurify (`http://www.webpurify.com/`) can also be used - it is much more versatile than PurgoMalum, as it can moderate images and videos as well, at a fee of course.

- **Font Sizes according to screen size** The trouble with font on webpages is usually flexibility and scalability when it comes to displaying your webpage headlines on different screen sizes. Herein lies the absolute marvel that is FitText (`http://fittextjs.com/`). This plugin is an open-source project available on github, that makes headline font sizes scalable to different screen types and sizes.

7 Technologies

Here is the technologies the client asked for and that we are going to use.

- Linux (Operating System)

- Git (Version Control System)

- JavaEE (Java Platform Enterprise Edition)

- JPA (Java Persistence AP)

- JPQL (Java Persistence query language)

- JSF (Java Server Faces)

- HTML (Hypertext Mark-up Language)

- AJAX (Asynchronous JavaScript and XML)

- CSS (Cascading Style-sheet)

- Apache Maven (Software Project Management and Comprehension Tool)