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*
Name(s) Surname *u*********
Name(s) Surname *u*********

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

1 History

Date	Version	Description
04-03-2015	Version 0.1	Document Template created - Keagan
07-03-2015	Version 0.2	Added section 3.1.1 - Matthew

Contents

2 Introduction

2.1 Purpose

2.2 Background

3 Architecture Requirements

3.1 Access Channel Requirements

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.2 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.

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.

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.

3.3 Integration Requirements

The system must be integrated with the LDAP server to obtain user information.

3.4 Quality Requirements

- 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 .

3.5 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.

- Git (Version Control System): Git is good for group projects and information share. It can show the progress and the groups 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.

- 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.

- JPA (Java Persistence AP): 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.

- 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.
- 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.
- 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.
- 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.
- 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.

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

7 Access and integration channels

8 Technologies

9 Conclusion

10 References