UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

# COS301 Mini Project Functional Architecture Requirements

## Group Name: Group 7_a


Roger Tavares *1*0167324
Thinus Naude *1*3019602
Kabelo Kgwete *1*1247143
Sylvester Mpanganer *1*1241617
Maphuti Setati *1*2310043
Ruth Ojo *1*2042804
Axel Ind *1*2063178
Lindelo Mapumulo *1*2002862
Maria Qumalo *2*9461775


## Git repository link:

`https://github.com/thinusn/COS301MiniProjectArchitectureRequirements`

Final Version

March 9, 2015

# Contents

# 1 Introduction

This document was compiled by our group during our meetings and was produced as a whole by the team.

This document contains specifications of the software architecture requirements. This is the infrastructure upon which the application functionality will be developed. The following non-functional requirements are addressed in depth with supporting diagrams (when necessary):

- Quality requirements.
- Integration and access channel requirements.
- Architectural constraints
- Architectural patterns or styles
- Architectural tactics or strategies
- Use of reference architectures and frameworks
- Access and integration channels
- Technologies

# 2 Architecture requirements

## 2.1 Architectural scope

## 2.2 Critical quality requirements

### 2.2.1 Scalability

#### Description

Scalability is an essential aspect of a system and is the ability of a system to be easily enlarged in order to accommodate a growing amount of work.

#### Justification

As the Buzz System should allow for more than a million concurrent users, the system must be able to handle that.

#### Mechanism

1. Strategy:

   Scalability can be achieved by:

   - Clustering: using more resources by running many instances of the application over a cluster of servers or instances.
   - Load Balancing: Spread the systems load across time or across resources through scheduling or queueing. In the case that the limit for a server has been reached, a new instance will have to be created in order to handle the number of increasing requests. On the other hand, if the usage of a server is way below the capacity, the number of instances will have to be reduced.

     In terms of databases:

   - Caching: to ensure no duplication or repeated retrieval of frequent objects or queries; a separate module can facilitate caching; thus system resources will not be used up unnecessarily.

2. Architectural Pattern(s):

   - Blackboard Architectural Pattern
     - Makes used of a spaced-based architecture, making it easy to add a new knowledge source or update an existing knowledge source.

### 2.2.2 Security

#### Description

Security includes the securing of confidential data stored from unauthorized access and modification. For BuzzSpace, this means:

- Ensuring only genuine users may have access to the systems data.
- Different levels of access should be provided to users.
- Securing vulnerabilities that may exist in the application.

#### Justification

Security is a crucial aspect of the system. Poor security design and implementation can compromise the data stored by the system and hackers with malicious intentions can succeed to gain access to sensitive information. Using a combination of security strategies and patterns, we can prevent unauthorized users from accessing sensitive information or cause damage to the system.

#### Mechanism

1. Strategy:

   Security can be achieved by using:

   - Authentication: This is the process that ensures and confirms a users identity.
   - Encryption: This process converts sensitive data into another form, which cannot be easily understood by unauthorized users. Data will be converted to readable format for authorized users.
   - Input Validation: The process of validating all the input to an application before using it. This will prevent malicious attacks such as SQL injections.
   - Auditing and Logging: This includes the recording of the implementation level events and domain-level events that occur when the system is being used. This data will help identify what went wrong if the system experiences malicious attacks.

2. Architectural Pattern(s):

    - Authentication Enforcer pattern
        - This pattern can be used to manage and delegate authentication processes.
    - Secure Base Action pattern
        - Shows centralizing handling of security tasks such as input validation in a base action class.
    - Secure Logger pattern
        - This pattern can be used to log sensitive data and ensuring tamper-proof storage.
    - Secure Session Manager
        - This pattern can be used to show securely centralizing session information handling.

### 2.2.3 Usability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

### 2.2.4 Integrability

**Description:**

The Buzz system should be integrated with ease on any host site.

**Justification:**

The Buzz system is primarily designed for the CS website, but if in future a need to host it on another site arise,it needs to be easily integrated to the the site without any extra cost and the need to change the implementation of the system.

## Mechanism:

1. Strategy:
   The contracts based decoupling strategy can be used to implement integrability and achieve extensibility on the system by providing an interface between the system and the host site which will improves decoupling on the host site and the system.

2. Architectural Pattern:
   Messaging can be used to enhance decoupling and increase flexibility, the Buzz system or host site can change without affecting the other party, both parties will act as independent component that use an interface between them to request services and share resources.

## 2.3 Important quality requirements

### 2.3.1 Performance

#### Description

Performance is an indication of the responsiveness of a system to execute some action(s) within a specific time interval. This time interval is usually measured in terms of latency or throughput.

> **Latency:** the time it takes the system to respond to any event(s).
>
> **Throughput:** the number of events executed within a given amount of time by the system.

A system with high performance maximizes throughput and minimizes latency. This is ideally what we want for the BuzzSystem

#### Justification

We chose performance as an important quality requirement because the over/under performance of a system will influence other quality attributes of the system. If the system has increased latency and decreased throughput( for example due to the systems inability to handle increased load, or subpar scalability) the performance, responsiveness and usability of the system will suffer. Because the Buzz System will be used by users we want them to use a system the performs optimally and responds to their event(s) with minimal latency.

#### Mechanism

1. Strategy: The system needs to be responsive. The time it takes for a request to be sent back to a user should be minimized.
    - The system needs to be hardware and software fault tolerant (to a certain degree). The system needs to continue working/running as intended but possibly at a reduced level, rather than breaking/stopping completely. A system that is off-line has 0 throughput.
    - The UI presented to the user needs to be clean, dynamic and use minimal resources. For example using minified JavaScript files and compressed images. In order to reduce latency.

- The performance of the system needs to be optimal, memory or processor intensive tasks should run/execute when there are the least number of active users in order to minimize the impact these tasks will have on performance. For example run batch plagiarism and netetiquette checks at 2AM in the morning.
- The system needs to have a coping mechanism when there is a sudden change in its environment. For example if the system can currently handle 100 connections and suddenly there is 10000 connections. the system needs to know how to cope. Performance will suffer if this is not taken into consideration. See Scalability for possible solution.
- The system should deliver intermediate results or updates to the user when executing a request. For example, a web page that submits a form via AJAX can have a status/busy indicator to let the user know his/her request is being processed.
- When there is increased database server processing the system needs to keep latency low and throughput high. The system can achieve this by storing frequently requested database results or objects in a cache. This will ensure that frequent objects and queries aren't repeated or fetched over and over, wasting system resources and increasing database server processing.
- An important aspect of performance is to ensure the scalability of the system is optimal. Because an increase in performance directly affects the systems scalability, and in turn the lack of scalability also affects performance.

2. Architectural Pattern(s):

    - Many/all of the patterns discussed in this document could assist in performance.
    - Architectural Patterns that allow for scaling of resources are preferred.

### 2.3.2 Plug-ability (Maintainability)

#### Description

The system as a whole should be designed and developed in such a way that it is modular allowing for additional modules to be added or even removed.

#### Justification

The reason why this is an important quality requirement is because the system should allow the addition of new plug-ins(modules) or the removal of old modules that are no longer required or obsolete. This allows for a more adaptable system as it can be adjusted to a specific users needs. By ensuring plug-ability the system will also be much easier to test. Diagnostics of potential flaws and module clashes can now be eliminated as the system can be tested as individual modules and even as a whole allowing for multiple module configurations.

#### Mechanism

1. Strategy:
    - There are multiple strategies that can ensure plug-ability one such strategy as mentioned above is to subdivide the system into separate interconnect-able modules or plug-ins.
    - Another strategy is to split the system into many services that communicate with one another to perform the required functional requirements.
2. Architectural Pattern(s): The best Architectural pattern to realize this requirement would be the micro-kernel, this is because this pattern allows for a plug-and-play infrastructure meaning that modules can easily be added or removed or even rolled back to previous versions without affecting other services on the system. This is done by using the Internal servers of the micro-kernel.

### 2.3.3 Monitor-ability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

## 2.4 Nice to have quality requirements

### 2.4.1 Reliability and Availability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

### 2.4.2 Testability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

## 2.5 Architectural constraints

# 3 Architectural patterns or styles

# 4 Architectural tactics or strategies

# 5 Use of reference architectures and frameworks

# 6 Access and integration channels

## 6.1 Access channels

### 6.1.1 Human access channels

### 6.1.2 System access channels

## 6.2 Integration channels

# 7 Technologies