BACHELOROPPGAVE:

Service architectures for educational purposes

FORFATTERE:

Arnt-Helge Nilsen Øyan
Stian Svalstad
Sigve Næss

DATO:

18.05.2016

Sammendrag av Bacheloroppgaven

Tittel:	**Tjenestearkiteurer til bruk i utdanning**
Dato:	18.05.2016
Deltakere:	Arnt-Helge Nilsen Øyan Stian Svalstad Sigve Næss
Veiledere:	Erik Hjelmås
Oppdragsgiver:	Norwegian University of Science and Technology
Kontaktperson:	Kyrre Begnum, kyrre.begnum@hioa.no
Nøkkelord:	Norway, Norsk
Antall sider:	129
Antall vedlegg:	
Tilgjengelighet:	Åpen

Sammendrag: Flackr og Factory er to forskjellige tjenestearkutekturer som kjører på moderne og relevante teknologier. Studentene vil ta på seg rollen som systemadministrator og ha ansvaret for systemdrift. De vil få en teoretisk og praktisk innsikt i relevante og reelle IT temaer som skalering, tilgjengelighet, single point of failure, og sikkerhet. Som en del av dette skal de installere, konfigurer og drifte nettjenere, database, og servere som gjør lastbalansering av inkommende trafikk.

Summary of Graduate Project

Title:	**Service architectures for educational purposes**
Date:	18.05.2016
Participants:	Arnt-Helge Nilsen Øyan Stian Svalstad Sigve Næss
Supervisor:	Erik Hjelmås
Employer:	Norwegian University of Science and Technology
Contact Person:	Kyrre Begnum, kyrre.begnum@hioa.no
Keywords:	Service-oriented architecture, MEAN, stack, architecture, design, RabbitMQ, message queue, web application, IT operations
Pages:	129
Attachments:	
Availability:	Open

Abstract:

Flackr and Factory are part of two different service architectures running on modern and relevant technologies. The students will take on the role as a system administrator and be responsible for system operation. They will get a theoretical and practical insight on relevant and real-world IT operational issues like scalability, availability, single point of failure, and security. As a part of this they have to install, configure and operate web servers, databases, and servers doing load balancing of the incoming traffic.

Preface

This bachelor thesis has been an interesting and challenging project. During the time working we have learned different technologies and better understand how they are installed, configured and used at companies. Spending time researching, finding specific information and figuring out how to design a working system architecture has been something to look back at. We really enjoyed taking IMT3441 database and application administration when we took the course, and we hope this will contribute to making the course even better in the future.

We would like to thank Erik Hjelmås for being our supervisor, keeping us engaged by asking relevant questions and providing us with feedback throughout our bachelors project. Kyrre Begnum for sending in the suggestion, his insight, our productive discussions, and of course valuable feedback. Michael Behrns for helping with HTML and CSS.

Contents

Preface . . . . **iii**
Contents . . . . **iv**
List of Figures . . . . **viii**
List of Tables . . . . **ix**
1 Introduction **1**
1.1 Project background 1
1.2 Project description 1
1.3 Target Audience 2
1.4 Goal 2
1.5 Academic Background 2
1.6 Development Model 2
1.7 Document Structure 4
1.8 Terminology 5
2 Requirement Specification **6**
2.1 Functionality 6
2.2 Usability 6
2.3 Reliability 7
2.4 Security 7
2.4.1 Web Application Security 7
2.4.2 Architecture Security 8
3 First Architecture - Flackr **9**
3.1 Technology 9
3.1.1 MEAN Stack 9
MongoDB 9
Express.js 9
Angular.js 9
Node.js 10
3.1.2 npm 10
3.1.3 Mongoose 10
3.1.4 Jade 10
3.1.5 fs 10
3.1.6 PM2 10
3.1.7 Nginx 11
3.1.8 GlusterFS 12
3.2 Design 13
3.2.1 Application 13
Node.js 14
Jade 14
Express.js 15
Mongoose 15

PM2 . . . 16
3.2.2 Architecture . . . 16
Nginx . . . 17
Application server . . . 17
MongoDB . . . 17
GlusterFS . . . 18
3.3 Implementation . . . 18
3.3.1 Application . . . 18
Node.js and Express.js . . . 18
3.3.2 Architecture . . . 26
Node.js and Express.js . . . 26
MongoDB . . . 27
Mongoose . . . 30
PM2 . . . 30
Nginx . . . 31
GlusterFS . . . 32
3.4 Traffic generation . . . 34
3.4.1 Introduction . . . 34
3.4.2 Implementation . . . 34
3.5 MEAN Stack Security . . . 38
3.5.1 MongoDB . . . 38
3.5.2 Express.js . . . 39
3.5.3 Angular.js . . . 39
3.5.4 Node.js . . . 39
3.6 Testing . . . 40
3.6.1 System testing . . . 40
3.6.2 Static code analysis . . . 40
3.6.3 Performance testing . . . 41
3.6.4 User manual review . . . 41
3.6.5 Traffic generation script testing . . . 41
4 Second Architecture - Factory . . . 42
4.1 Technology . . . 42
4.1.1 RabbitMQ . . . 42
4.1.2 Flask . . . 42
4.1.3 Pika . . . 42
4.1.4 PM2 . . . 42
4.1.5 MongoDB . . . 42
4.1.6 Munin with RabbitMQ plugin . . . 43
4.1.7 RabbitMQ Management Plugin . . . 43
4.2 Design . . . 43
4.2.1 Application . . . 43
Entrypoint . . . 43
Workers and queues . . . 43
Safe . . . 43
Finished Job Entry System . . . 43
Common . . . 44

4.2.2 Architecture . . . 44
4.2.3 Monitoring . . . 44
Munin with RabbitMQ Plugin . . . 45
RabbitMQ Management Plugin . . . 45
4.3 Implementation . . . 45
4.3.1 Application . . . 45
Entrypoint . . . 45
Workers . . . 45
Safe . . . 47
Finished Job Entry System . . . 49
4.3.2 Architecture . . . 49
RabbitMQ . . . 49
Common . . . 50
MongoDB . . . 50
4.3.3 Monitoring . . . 50
Munin with RabbitMQ Plugin . . . 50
RabbitMQ Management Plugin . . . 52
4.4 Traffic generation . . . 52
4.4.1 Introduction . . . 52
4.4.2 Implementation . . . 53
4.5 RabbitMQ Security . . . 54
4.5.1 Access Control . . . 54
4.5.2 Messaging transport security . . . 54
4.5.3 RabbitMQ Security Checklist . . . 54
4.6 Testing . . . 54
4.6.1 System testing . . . 55
4.6.2 Static code analysis . . . 55
4.6.3 Performance testing . . . 55
4.6.4 User manual review . . . 55
4.6.5 Traffic generation script testing . . . 55
5 Discussion . . . 56
5.1 Results . . . 56
5.1.1 Project outcome . . . 56
5.1.2 What did we not do . . . 56
5.1.3 What could have been done differently . . . 56
5.1.4 Technologies . . . 57
5.1.5 Time usage . . . 57
5.1.6 Complications . . . 57
5.2 Group evaluation . . . 58
5.3 Further Development . . . 59
6 Conclusion . . . 60
Bibliography . . . 61
A Application code and configuration files . . . 64
A.1 Flackr . . . 64
A.2 Factory . . . 71
B Traffic generating script for Flackr . . . 80

C Traffic generating script for Factory **86**
D Disable Transparent Huge Pages **89**
E MongoDB configuration file **90**
F Nginx configuration file **91**
G User manuals **92**
G.1 Flackr 92
G.2 Factory 94
H Verification from Kyrre **99**
I A JSON example from https://randomuser.me **100**
J Meeting logs **101**
K Pre plan document **110**
L Gantt **129**

List of Figures

1 Iterative development model 3
2 Nginx Reverse Proxy with Load Balancing 11
3 GlusterFS Design 12
4 Deployment view of Flackr 16
5 Deployment view of Factory 44
6 Gantt chart 129

List of Tables

1 Table with versions 14
2 Image scheme 15
3 Comment scheme 16
4 Comparing SQL and NoSQL 18

1 Introduction

1.1 Project background

In IMT3441 database and application administration at Norwegian University of Science and Technology you get introduced to one of the more traditional service architectures by implementing a web application called Bookface, a really simplified social network. This architecture consists of the Linux Operating System, Apache HTTP Server, MySQL Database and PHP for web development. This is more commonly known as the LAMP stack. The students will be tasked with installing, configuring and operating this application as it gets hammered with traffic. Implementing load balancing between several web servers, caching technology on application level and much more in order to maintain continuous operation.

Lately, it has been more relevant to provide an alternatives and more modern architectures in addition to the LAMP stack, in order for the student to benefit more from the course. Examples would be a Node.js solution with API calls, unstructured databases, or a gaming application service. This will be in addition to Bookface, in order to let the students either choose, or operate them all.

1.2 Project description

This project deals with the design and implementation of alternative architectures. A minimum of two complete architectures are expected to be delivered at the end of this bachelor thesis. Hence, this project will be divided into two parts, one for each architecture.

Each architecture will be used in IMT3441 Database and application administration. However, it is expected that one of the architectures is of similar size as the current Bookface application. Moreover, the other will be a smaller application to be used in a two week period during the course as an addition to the two larger service architectures. Furthermore, this project can be broken down into main objectives that is considered to be the same for both architectures.

- Decide upon a service architecture and install required software
- Design, develop and implement an application
- Develop a tool for generating traffic
- Write user manuals

In addition to the architecture itself, our employer also requires the following:

- A codebase in a git repository, where the versions can tell a "story". For example you start off with a version with several mistakes or errors by the developers. Either they didn't have time or resources to fix this before it had to be deployed
- Extra tools that are to be used from the "uptime" system, that directs traffic to the architecture and puts it under load, so that it feels more realistic. Much like the same way Bookface had users and posts added during the course
- Documentation. It can't be too complex and preferably be based on standard pack-

ages in Ubuntu. One must also take in to account the technical expertise of the persons taking the course. If there is too much hacking, it might overshadow the big picture

There is no equivalent course given today. This bachelor project will introduce an unique value for everyone who wants to learn more about operating large scale systems. Furthermore, since the documentation is written in English, the possibility of offering this to international students in the future will be achievable. Since the LAMP-stack is already implemented in the current course, our solution will be based on another stack.

1.3 Target Audience

Our target audience is first of all our employer, however, as he is intending to use it in one of his courses, the students are part of our target audience as well.

1.4 Goal

Our current goals for this project are as follows:

Learning

- Be able to understand and differentiate between different service architectures
- Learn to install, configure and implement software solutions
- Be able to design documentation that enhances quality of learning
- Further develop knowledge of relevant professional areas like programming, network, scripting and, databases
- Implement the use of "Best practices"

Performance

- Design two or more service-oriented architectures to compliment the current Bookface
- Write documentation so students can configure, install and operate the service architectures
- Develop a method to send traffic to the service architecture

1.5 Academic Background

Our group consists of three students from two different study programs. Arnt-Helge Nilsen Øyan and Stian Svalstad are studying Science in Network and System Administration. Sigve Næss is studying information security. Both courses are now combined in to IT Operations and Information Security. All of us are third year students at NTNU in Gjøvik (former Høgskolen i Gjøvik).

1.6 Development Model

Being able to add, remove or change previous steps during the design or development is a major advantage, therefore an agile method would be the best fit for our project. It was also necessary for us to be able to store, and keep track of possible features that could be implemented at different stages of the project period. A solution to this would be a product backlog, which is often used in agile methods. The way IMT3441 works is you start by installing a webserver, put Bookface source files in the web root folder, install MySQL servers, and connect them together. This is considered to be an incremental way of working, which we acknowledge as a favourable method for developing system architectures.

Depending on how our application will look, there will most likely be some development or scripting. This largely depends on the outcome of our ideas. Therefore we need our model to be flexible enough to handle the unpredictability. Based on this and aspects earlier in the thesis we ended up with a incremental and iterative model [1]. The development method is illustrated in figure 1.

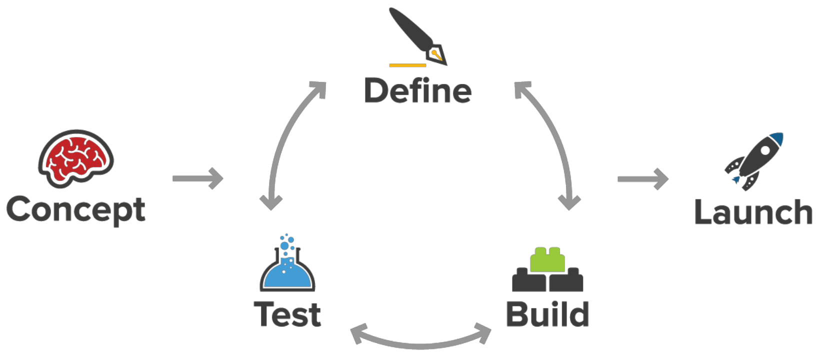


Figure 1: Iterative development model

1.7 Document Structure

This project document is structured into the following sections:

1. Introduction: A brief introduction to the project as a whole, background of the project and the planning process.
2. Requirement Specification: A document that describes how the system is expected to perform and which precautionary methods has been considered.
3. First Architecture:
    - Technology: A brief description of the different components and technologies used in the first architecture.
    - Design: Detailed description of the architecture and system design, and how the different technologies are combined.
    - Implementation: Documented details about the implementation of the system, and explanation of decisions taken related to the implementation.
    - Traffic Generation: Contains information and description about the tool that developed to generate traffic in this system.
    - Security: An architecture specific section that contains security related information and recommendations about the relevant technologies used in this architecture.
    - Testing: Description of how the system has been tested.
4. Second Architecture:
    - Technology: A brief description of the different components and technologies used in the first architecture.
    - Design: Detailed description of the architecture and system design, and how the different technologies are combined.
    - Implementation: Documented details about the implementation of the system, and explanation of decisions taken related to the implementation.
    - Traffic Generation: Contains information and description about the tool that developed to generate traffic in this system.
    - Security: An architecture specific section that contains security related information and recommendations about the relevant technologies used in this architecture.
    - Testing: Description of how the system has been tested.
5. Discussion and Conclusion: Discussion of the results and a final evaluation of the project and system.
6. Appendix All appendices.

1.8 Terminology

JS JavaScript programming language
JSON JavaScript Object Notation
LAMP Linux, Apache, MySQL, and PHP/Python/Perl
MEAN MongoDB, Express, Angular, Node.js
URI Uniform Resource Identifier
URL Uniform Resource Locator
THP Transparent Huge Pages
TLB Translation Lookasie Buffer
PPA Personal Package Archive
CSS Cascading Style Sheets
HTML HyperText Markup Language
pip Package manager for Python.
npm Node package manager
ODM Object Data Manager
TLS Transport Layer Security
SSL Secure Socket Layer
OWASP Open Web Application Security Project
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
SMTP Simple Mail Transfer Protocol
POP3 Post Office Protocol v3
IMAP Internet Message Access Protocol
UQL Unstructured Query Language
LTS Long Term Support
PID Process identifier
GUI Graphical User Interface
CLI Command Line Interface
YAML Human readable data serialization language

2 Requirement Specification

In this chapter the requirements for our architectures are discussed. Some of the required functionality, as well as the specifications for usability and reliability are described.

2.1 Functionality

In terms of functionality there are certain requirements we have to comply with, as to what is expected from our employer. First of all, we distinguish between the two architectures. The first architecture is expected to be of similar size as the current Bookface stack, hence it has to include an application. This means that the required functionality will have to include methods for accepting new entries as well as being able to display them on a simple website. For it to be usable for our employer in one of his courses, the application needs to be created in such a way that we can use scripting methods to generate traffic to the application, in order for them to seem dynamic and authentic. In terms of the architecture itself, and not the application, key features such as load balancing, support for scalability and storage methods including the use of a database, are all key concepts that are expected regarding functionality. Whereas, the first architecture is intended to be used alongside Bookface, the second architecture has a different area of use. The intention of our employer is to use it at the end of his course to illustrate operational issues, more about this later on. With this in mind our second architecture has fewer of these concepts as requirements. However, the application still has to accept data and display it in a representative way.

The traffic generating scripts for both architectures are required to have similar functionality, this is due to the fact that our employer will use both scripts the same way. Accordingly the method of execution for each script will be with command line arguments for an IP address as well as the necessary options. The options will be different for each script, these options are discussed in extent in the traffic generation section for each respective architecture.

2.2 Usability

As specified in the task description, everything that needs to be installed has to be based on native packages in Ubuntu, except the Git based code repositories. This is due to the simple complexity that is required from our architectures. The installation has to be simple enough for students with little to no experience to be able to install and configure the systems. This means that the user manuals that we write will have to be accurate and explicit, in order to ensure that the applications can be installed without implications. In terms of our traffic generating scripts, the code has to be easily readable as well as easily executable. Accordingly, we are following the Google Python Style Guide[2].

2.3 Reliability

In terms of reliability, it is a requirement that the solutions work with 14.04 Ubuntu server LTS. However, the architectures should be created in such a way that it can be easily implemented and used with newer versions of Ubuntu server, without major changes. It is expected that the solutions are able to operate for a time period of a school semester.

2.4 Security

Due to the requirements and circumstances of our bachelor thesis, our main focus will be on architecture administration and operations, with less focus on the security aspect. If we were to deploy our architectures on the public Internet we would have to put a lot more effort and hours into the security of our systems.

Although our focus primarily will not be on security, it does not mean that we have completely overlooked security. In each architecture there will be a part that focuses on security. But first we will go through some basic concepts of architecture and application security, which are relevant for our architectures.

2.4.1 Web Application Security

The first architecture will be a web application, and therefore looking into this kind of security is relevant and important for us. Web application security is a major part of information security that deals specifically with securing web applications, websites and web services. Security measures should be applied continuously throughout the application lifecycle to guarantee a secure application environment.

Due to the increasing security risks involved in Web Application Security, OWASP have created several guides, cheat sheets and other documents to help organizations secure their web applications.
In 2013 OWASP created this list of the top 10 critical web application risks[3]:

1. Injection
2. Broken Authentication and Session
3. Cross-Site Scripting(XSS)
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forger(CSRF)
9. Using Known Vulnerable Components
10. Unvalidated Redirects and Forwards

If the web application was meant to be available on the open web, it would have been recommended to go through a guide like the OWASP Application Security Guide[4] before a potential official release of the application.

2.4.2 Architecture Security

Architecture Security is a type of security design that covers potential risks involved in a certain scenario or environment. It also describes where the security controls/countermeasures are positioned, and how they relate to the system architecture. These security controls/countermeasures are in place to maintain the architecture and system quality such as confidentiality, integrity and availability.

It is important that we take security of each architecture component into consideration while designing the architecture to ensure a secure environment. Architectures must therefore be structured in a proper way so that appropriate security controls may easily be implemented.

3 First Architecture - Flackr

The first part of this project will cover the development of an architecture based around a photo sharing web application. The whole idea is based around the well-known and already existing application named Flickr. We figured out that this kind of application and it's architecture would fit our project in terms of requirements and design. Flickr is a popular web-based photo-sharing and hosting service with advanced and powerful features. It supports an active and engaged community where people share and explore each other's photos[5]. Our application is a very simple version of Flickr, and has the very creative name of Flackr. Since the core of our project is not focused around the front-end, the application has limited user functionality. The next sections will discuss the whole project process for Flackr, which is made up of Technology, Design, Implementation, Traffic Generation, Security, and Testing.

3.1 Technology

First off in this section, all the modules, applications and other technologies that are required to make a complete deployment of Flackr are discussed. The idea of this chapter is to get a basic understanding of how the different technologies work. Specific descriptions on how the different technologies are used in this project will be presented in Design, which is the next chapter.

3.1.1 MEAN Stack

MEAN is a free and open-source JavaScript software stack for building dynamic web sites and web applications. The MEAN stack makes use of MongoDB, Express.js, Angular.js, and Node.js. Because all components of the MEAN stack support programs written in JavaScript, MEAN applications can be written in one language for both server-side and client-side execution environments[6].

The **MEAN** stack consists of the following components: **M**ongoDB, **E**xpress.js, **A**ngular.js and **N**ode.js.

MongoDB

MongoDB is the leading open source NoSQL database, which uses a document-oriented data model.

Express.js

Express.js is a minimal and flexible Node.js web application framework, providing a robust set of features for building single and multi-page, and hybrid web applications.

Angular.js

Angular.js makes it possible to extend HTML vocabulary for an application. But since the frontend of the application will not be used by real users, simple html will be used instead.

Node.js

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.

3.1.2 npm

npm is the default package manager for the JavaScript environment Node.js. npm was first created to make it easy for developers to manage dependencies for their JavaScript applications. Most of the software, modules and packages used in this architecture is based on JavaScript, and is not available on the more traditional package managers such as APT.

3.1.3 Mongoose

Mongoose is an ODM library that translates data from the database to JavaScript objects that can be used in the application. In this case the most useful part of mongoose is the possibility to make structured schemes. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

3.1.4 Jade

Jade is a template language, which compiles to HTML. Jade makes your HTML code very clean and easy to implement. There are many alternatives to Jade, but since Jade comes as default in Express.js we decided to stay with it.

3.1.5 fs

The Node.js filesystem is a module that consist of over 50 different functions. The two main functions in fs is reading and writing to a file.

3.1.6 PM2

PM2 is an advanced, production process manager for Node.js applications. PM2 is used to keep your application in production at all times. It is able to control the state of the application, by spawning a PM2 daemon. It can launch the application at system boot, monitor it and log the application activities.

3.1.7 Nginx

Nginx is a web server that also works as a reserve proxy server for HTTP, HTTPS, SMTP, POP3 and IMAP protocols. A reverse proxy server is a type of proxy server that directs requests from clients to the appropriate back-end server.

Nginx can also be used as a load balancer. Load balancing is a commonly used technique for optimizing resource utilization, maximizing throughput, reducing latency, and ensuring fault-tolerant configurations[7]. Load balancing helps distribute traffic to several servers, which improves performance, scalability and reliability of the running application.

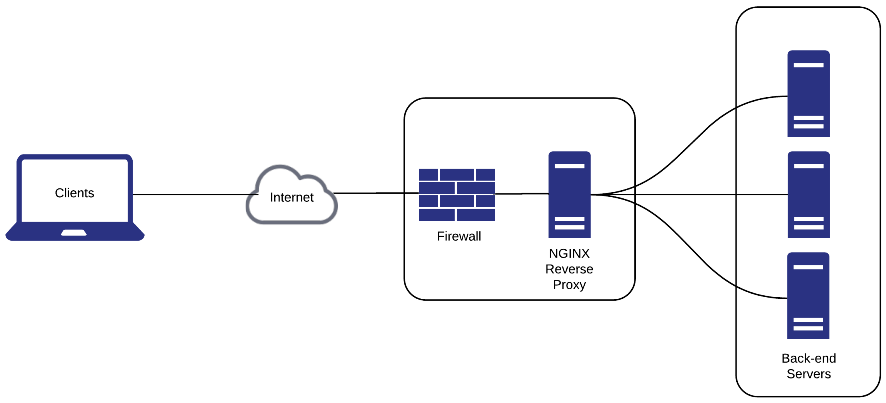


Figure 2: Nginx Reverse Proxy with Load Balancing

3.1.8 GlusterFS

GlusterFS is a distributed scalable network filesystem. A distributed filesystem is a method of storing and accessing files based in a client/server architecture. This means that the file system will allow clients to access and process data stored on the server as if it were on their own computer.

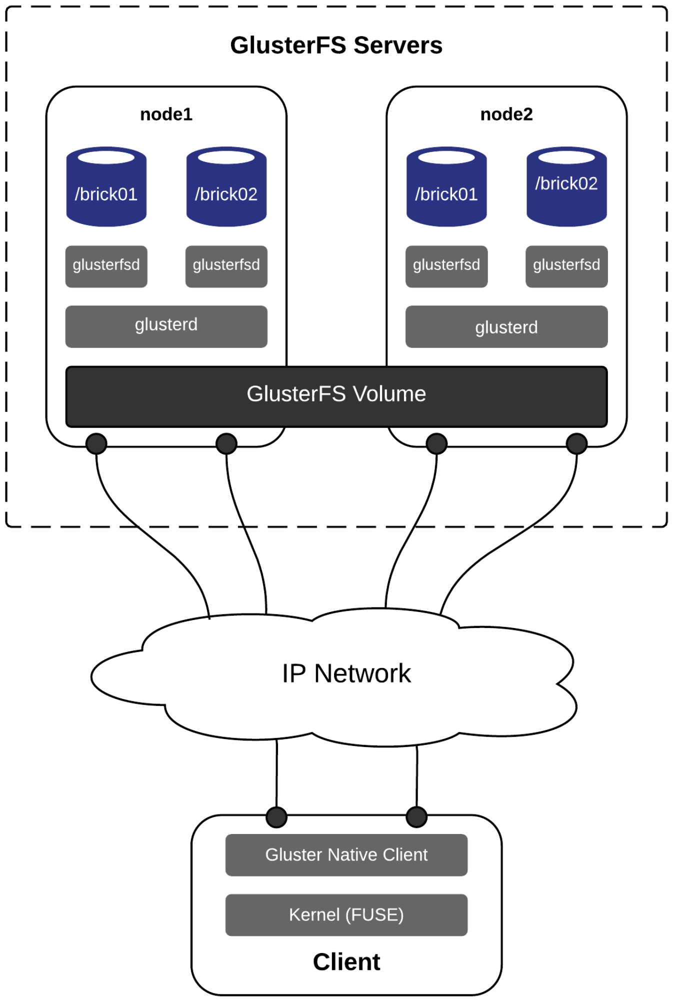


Figure 3: GlusterFS Design

3.2 Design

This chapter will dig into how the technologies that were discussed earlier work together and how they are a part of the architecture. Our thought process and design solutions are explicitly discussed. This section will be split in two. First the reader will become familiar with how the application is built. The second part will cover how the architecture behind it works in order to support the application.

Generally, all servers will run on Ubuntu Server 14.04.3 (Trusty Tahr) amd64 image available in SkyHiGh. SkyHiGh is the university's OpenStack solution and is used in IMT3441.

3.2.1 Application

Initially we tried to look around to see if it was possible find an existing web application that could run on the architecture we had in mind. The few already existing web applications that we found were either too complex, difficult to see how the components are working together, or would most likely not be viable in the future. Therefore we decided to develop our own application that would fit this project.

We didn't think it was necessary for the employer and course coordinator to make changes to the curriculum every semester the course is held, because of new releases. The first version should be unpolished. As the students deploy newer versions, the stability will improve and the code is revised.

As mentioned in the requirement specifications, this application will be available in three different versions. The details can be seen in table 1. Version two and three has implemented dark launch. The idea is to release a new feature, but keep it kind of hidden. For example in the background the application can do queries every time a user does a certain task, but the result of this query is hidden. This way you can test certain features as close to a production environment without presenting the result to a user. Companies like Facebook have used this to stress test their own infrastructure[8].[9]

Version	Description
1	• Initial setup • All entries on front page • 1 application server • Random logging • PM2
2	• Limit the number of entries on front page • 1 application server • Dark launch: Database query for the most viewed image
3	• GlusterFS replication between application servers • Nginx load balance, cache and reverse proxy • 2-3 application servers • Simple implementation of dark launch, query result will now be shown on the front page

Table 1: Table with versions

Node.js

In this architecture, Node.js is what Apache is to the traditional LAMP. It is an asynchronous HTTP server. Most of the code here is written by Express-generator module that creates an application skeleton.

Jade

Jade is a template engine primarily used on server side to render templates in Node.js. This way we can have static template files and fill them with variables at runtime. Currently there are two different templates, index and a single image page based on ID. They both have a common layout template they build from.

An example snippet of our layout looks like this in HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <link rel="stylesheet" href="/stylesheets/style.css">
    <div id="logo">
      <a href="/">
        <img src="/images/flackr_logo.png" alt="Flackr Logo">
      </a>
    </div>
    <h2>The #1 website for sharing your images!</h2>
  </head>
  <body>
  </body>
</html>
```

This is how it looks like in using a jade template

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
    #logo
      a(href='/')
        img(src='/images/flackr_logo.png', alt="Flackr Logo")
    h2 The #1 website for sharing your images!
  body
    block content
```

Express.js

Express.js is a framework for Node.js, which is used to provide features for web and mobile applications [10]. In this project, a Route Listings structure [11] with Router class [12] will be used in order to create modular and mountable route handlers. This is a complete middleware and routing system that will handle all requests.

The application have two pluggable views, which is the index site / and insert /insert. Index presents a matrix with images and a single image page based on ID. Insert is used for inserting either a new image or comment. All routes will respond with a template site. Flackr have custom templates for 200 OK and 404 Not Found.

Mongoose

In this architecture Mongoose is used as an abstraction layer instead of using the native driver for Node.js. By using Mongoose it is possible to abstract away named collections of arbitrary objects, default values and it also has built in validation.

Mongoose uses a scheme to structure the data. This results in consistent schemes across our database for every entry that gets inserted. Two different schemes will be used, where the first one is for an image and the second is for comments. A single image can have several comments.

Attribute	**Description**
ID	Unique identifier. We also use this to find the image connected to the post
Title	Post title
Publisher	Name of the publisher
Views	Incremental view count
Comments	Array of comments
Date	The time when the image was published on Flackr

Table 2: Image scheme

Attribute	Description
Name	Name of who posted the comment
Text	Comment
Date	The time the comment was posted on Flackr

Table 3: Comment scheme

PM2

It is highly recommended using a process manager to run Node.js applications, and therefore PM2 will be used. Advantages of using PM2 includes easier management, performance monitoring and overview of resource consumption while running.

3.2.2 Architecture

The architecture needs to be simple and relevant. We started off by researching Web Application Hosting in the Amazon Reference Architecture to plan the design. Web hosting can be complex with wild swings in traffic pattern. Reliablility, scalability, security and high performance are key words here [13]. Figure 4 is a concept design which represents the architecture as a whole.

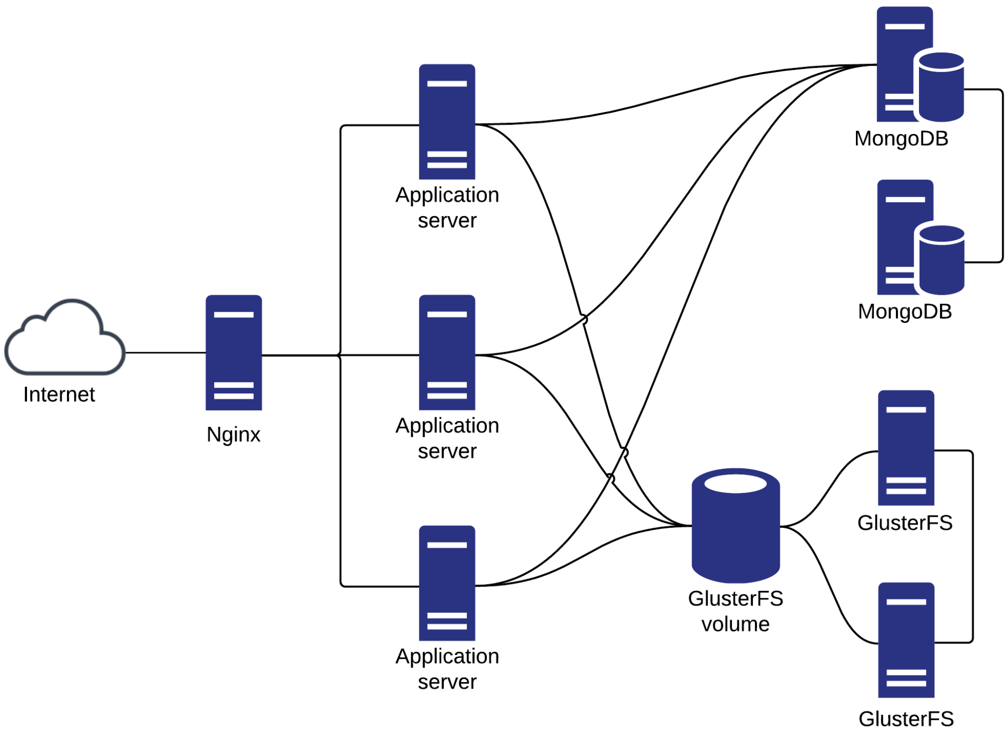


Figure 4: Deployment view of Flackr

Nginx

The point of entry into our application will be a server running Nginx, which has three roles:

- Reverse proxy
- Load balancer
- Caching

There are several reasons why it is recommended to implement this in an architecture:

- Simplifying port assignments for multiple applications
- Increased reliability in case the application crashes
- Increased security by not letting the application directly connect to the Internet, exposing possible open ports and preventing security flaws in one of the technologies the application is using [14]
- Easier scaling with adding only the IP of the application server to the configuration file

Load balancing with Nginx is easy to configure, and has a ton of features such as session persistence, weighting slower servers, limiting connections, and health monitoring to mention a few. This way it is possible to create a dynamical environment and tailor it to meet the requirements needed. The student should be free to configure this themselves, but a general configuration example will be available in appendix F.

In order to get the most out of the application, caching will be implemented. This way it is possible to serve static content and lessen the traffic load on the application servers. As well as serving static content, a 5 second cache of the front page will be implemented. This is called microcaching[15] and it is used by sites like Imgur[16].

Application server

The application server is where Node.js environment, Node.js application and PM2 will be installed and configured. There will not be a limit on these kind of servers, and therefore it is possible to scale and meet the demands from users.

MongoDB

Each image has its own entry in the database that consists of the data mentioned in tables 2 and 3. This data will be stored using MongoDB, a NoSQL database. Compared to MySQL, a traditional relational database. MySQL uses tables with certain keys to define the relation, in a pre-defined database schema, and uses SQL for database access. MSSQL and SQLite are other SQL implementations. In MongoDB the tables are JSON-like documents with dynamic schemes and key-pair values. There are no need to define the structure of the document, such as fields or types of values. The query languages are both rich, powerful and portable so there are no need to send multiple commands to fetch the desired data. Other than MongoDB, Redis, Cassandra and CouchDB are other alternatives of NoSQL implementations[17]. Table 4 is a summary of the comparison.

SQL	NoSQL
Relation database	Non-relational
Tables with rows and columns	Documents of key-value pair
Predefined schema	Dynamic schema
Vertically scaleable by increasing the hardware	Horizontal by increasing the number of database in the pool
SQL for defining and manipulating the data	Queries are focused on collection of documents, sometimes called UQL
MySQL, MSSQL, SQLite	MongoDB, Redis, Cassandra, CouchDB

Table 4: Comparing SQL and NoSQL

GlusterFS

Flackr will store all images in a file system within the operating system . In order for Flackr to be scalable, it was required to find a way to replicate the images over several servers. In IMT3441 students are introduced to GlusterFS, therefore this kind of technology will be used instead of using object storage or using a rsync/cronjob. By using GlusterFS it will provide the architecture with fault tolerance and load balancing.

The GlusterFS setup in this architecture will consist of two GlusterFS servers with a cinder volume attached. It was considered having several volumes attached, but this is not supported in OpenStack at this moment [18]. The files will be accessed using GlusterFS-client. This way the application servers have a mounted share where all the images are replicated, so it does not matter which one gets the request.

3.3 Implementation

Implementation is the chapter where the installation process of our stack and application is explained. It goes into specific details regarding commands, setup and written code. This chapter is also considered to be the foundation for the user manual, which is added as appendix G.1

As already mentioned, it is assumed that students know how to launch servers in SkyHiGh, and connect through ssh.

3.3.1 Application

This section describes the process behind the creation of the Flackr application.

Node.js and Express.js

In a later section it is discussed how we started up with Node.js and Express.js, using a template generator that creates a bunch of files required for a basic Node.js application. Furthermore, some of these files are discussed in this section as well as certain important aspects of the application.

Before discussing any of the code, it is briefly explained how the Express routing and route methods work. Routing is the application end points (URIs) and decides how the application should respond to the requests. This is done by attaching a HTTP request method to an instance of the express class along with a path and handler. A path can be the home-, customer, or information page on the website. Handler is the function executed when the route is matched, typically a request (req) and response (res) object. Request have the headers, body, and a query string. Response is the object being returned when the application gets a request. This could be a simple HTTP status code, a query string, and a template object. In Flackr, we're using this along a router object. This creates isolated instances of routing. This makes the application more structured, orderly, and more module based. The modules can then be enabled to a particular root URL in the app file for our project. For example, we want to route all of '/hello' to the file 'world'. This way we could access the page using '/hello/world'.

In the file 'application-name/routes/world.js'

```
router.get('/world', function(req, res, next) {
  res.send('Hello back at you.')
}
```

And then 'app.js', the main file.

```
app.use('/hello', world);
```

First of all certain changes have to be made to the app.js file. This file is the core of the application, it is the file that controls everything. When the application is started, app.js is the starting point. Express-generator has created a basic structure of the file. Which has been edited and more configuration has been added to it. As the file is rather large, it is included as a whole in appendix A.1.1. and only the most important syntax is discussed here.

The app.js file deals with the connection to the database, this section of the file is discussed later on. Express needs to be initialized with the syntax

```
var app = express();
```

The application endpoints are defined

```
var routes = require('./routes/index');
var insert = require('./routes/insert');
```

This tells the app to load these files when the application launches, they will be used later on in the file. Next it is specified which rendering engine should be used, in this case it is jade.

```
app.set('view engine', 'jade');
```

Considering that the filesystem is used for image storing, in combination with GlusterFS in version 3. Therefore a variable is declared so that the application knows where they are stored.

```
app.use(express.static(config.photos.folder));
```

When express.static is used with a folder location, it is now possible to access the photos without using a full path. Next, it is defined which addresses are to be associated with which route. Now the ip address of the server + a '/' ending will do the routing located in the 'routes/index' file, and the '/insert' will do the routing located at 'routes/insert'.

```
app.use('/', routes);
app.use('/insert'. insert);
```

The last element that has been modified, was to configure which port the application will be listening on. We used either 3000 or 4000 when developing.

```
app.listen(4000, function(err) {
  console.error('press CTRL+C to exit');
});
```

The remaining of the file are handlers for development and production, including a general 404 not found handler. Before the routes and views are discussed, we have created a configuration file to specify environment variables such as usernames and IP addresses [19].

```
var config = {};

config.mongodb = {};
config.image = {};

config.mongodb.ip = '';
config.mongodb.name = '';

config.mongodb.username = '';
config.mongodb.password = '';

config.image.frontpagelimit = 20;

config.image.folder = ''

config.image.topviews = 0;

module.exports = config;
```

The variables are fairly self explanatory. The variable config.image.topviews is a part of the dark launch concept mentioned earlier. The intention being that the application does a query to the database, but the results aren't being displayed on the website. This means that the students can test the backend load with a query to the database every time someone visits a page. This will be implemented in the second version as mentioned earlier and is enabled by changing the zero to one. Config.image.folder variable is a full path to a folder. Once the folder is created, change the ownership to whoever is running the application. This will most likely be the user 'ubuntu'.

```
mkdir -p /data/images
chown ubuntu:ubuntu /data/images
```

As already mentioned, Mongoose uses schemas to define a document standard for creating a collection in MongoDB. Hence, a schema for the application is defined. This is an extract from the imageSchema.js file. It shows how it is possible to define a schema that can be used, and accordingly all the entries in the database will have the same structure as the schema.

```
var mongoose = require('mongoose');
var Schema = mongoose.schema;

var Comment = new Schema({
    name        : String,
    text        : String,
    date        : {type: Date, default: Date.now }
});

var Image = new Schema({
    id          : {type: Number, unique: true},
    title       : String,
    publisher    : String,
    views        : Number,
    comments     : [Comment], // array of comment schemas
    date         : {type: Date, default: Date.now}
});
```

It is now possible to require the ImageSchema file in our app.js file, as shown below.

```
var Image = require('./imageSchema');
```

Mongoose has the ability to have embedded documents, meaning that a schema can have an array of another schema. This makes it possible to have a single Image with many comments. As shown in the example above. There will now be a new document for each comment in the database.
To be able to use the database, a connection has to be established. In the application we are using a simple configuration file, in which the students will have to specify the appropriate IP address and database name. If the database doesn't exist, Mongoose will create it. Mongoose handles the connection with the following syntax, extracted from our app.js. Note that the error handling function has been removed for readability. First of all it requires the package Mongoose, as it is used to connect to the database. All of the environment variables are taken from the configuration file discussed earlier.

```
var mongoose = require('mongoose');
var config = require('./config');

connection = ('mongodb://' + (config.mongodb.username) + ':' + (
    config.mongodb.password) + '@' + (config.mongodb.ip) + '/' +
     (config.mongodb.name));
console.log(connection);

var options = {
  auth: {authdb: "admin"}
};

mongoose.connect(connection, options , function(err) {
  if (err) {
    console.log('Error connecting to database', err);
  } else {
    console.log('Connected to database');
  }
```

Earlier, it is defined that the app.js should look for a file named routes/index.js to handle the routing for '/' endings, this includes the handling for the front page and each individual site for each picture. The following code is the routing for the front page. The first part is the query for our dark launch feature flag, it finds the image with most views from the database and saves it in a variable, all depending on whether the feature flag is enabled.

```
/* GET picture */
router.get('/', function(req, res, next) {
  // If feature flag is enabled, find image with the most views
  if (config.image.topviews == 1) {
    Image.findOne({}).sort({views: -1}).exec(function(err,
        topImage) {
```

```
      if (err) {
        console.log('Error was thrown...');
        throw (err);
      }
      else {
        console.log("Most views ID: " + topImage.id);
        req.topImage = topImage;
      }
    });
  }
  else {
    req.topImage = null;
  }
```

Regardless of whether the feature flag is enabled or not, the application has to get the images for the front page and their associated data. It also limits the amount of pictures it returns with the config.image.frontpagelimit from the configuration file. This would give the students the opportunity to control or limit the amount of data requested. After the query has been completed, assuming an error wasn't thrown, it renders the requested data onto the handler for the views, located at views/index.jade, which we will discuss later on.

```
// Find top based on frontpagelimit and render it
  Image.find({}).sort({date: -1}).limit(config.image.
      frontpagelimit).exec(function(err, images) {
    if (err || !images.length) {
      res.status(404);
      res.render('error', {
        title : 'flackr',
        message : 'Object returned from database is empty.'
      });
    }
    else {
      res.render('index', {
        title : 'flackr',
        images : images,
        lastID : images[0].id,
        topView : req.topImage
      });
    }
  });
```

The function for handling a single image is similar to finding all of them, the difference being that it uses the Mongoose query findOne() instead of find(). The error handling code has been omitted here, but it can be viewed in the appendix. Assuming it didn't throw an error, it increments the image variable views with one, saves it, and renders the rest to views/image.jade.

```
/* GET :id */
router.get('/:id', function(req, res, next) {
              // Find one image based on route
    Image.findOne({id: req.params.id}, function(err, oneImage) {
      ————Omitted code————
      // Increment if page is visited
      oneImage.views++;
      oneImage.save();

      res.render('image', {
        title : 'flackr',
        image : oneImage
      });
    }
  });
});
```

Before discussing the views files, the routing for inserting new images and data to the application is explained. In order for the traffic script to be able to insert new images and comments, the application requires an URL entry point that accepts image data. More about that in the traffic script section.

Insert.js is split into two sections, both using the HTTP GET method. One for adding a new image and a second for adding a new comment to an image. The code is split out to a function with a callback. By using callbacks we do not wait around for a function to finish. Therefore, it can keep on doing other things while waiting for it. In this case, this function is used to find an ID for the new image entry while still continuing to do other tasks like sending a request for the image. If we encounter any errors while trying to find an ID, nothing will be written to disk or commited to the database. The complete code can be seen in appendix A.1.3.

```
function handlerDatabaseQuery(callback) {
  // Mongoose query
  Image.findOne().sort({'id': -1}).exec(function(err, query) {
    // If error from database, return with error
    if(err) {
      console.log('Error occured');
      return;
    } else if (query === null) {
    // Special case; if database is empty we don't know the last
        insert ID
      var newId = 1;
      console.log('New ID: ' + newId);
      callback(newId);
    } else {
      // Error handling and special case, last is that we get an
          ID
      console.log('Last ID: ' + query.id);
      var newId = query.id+1;
      console.log('New ID: ' + newId);
      callback(newId)
    }
  });
}
```

Once the function returns the callback, an imagepath can be generated. This is the full path to the destination where the image will be written to. The following functions takes an imagepath, a response object from a request sent towards an image site and passes it along. Once it is done writing the image to disk, the size is returned with the callback. This function is called upon when request get a response from the image site.

```
 function handlerImageDownload(response, localPath, callback) {
    // Creates a stream to the path provided
    var writeStream = fs.createWriteStream(localPath);
    // Writing data
    response.pipe(writeStream);
    // When done writing data, check the size and send it back
    response.on('end', function() {
      fs.stat(localPath, function (err, stats) {
      size = stats['size'];
      callback(size);
      });
  });
};

————Omitted code————
    // Filepath concat
    var imagepath = config.image.folder + newId + '.jpg';
```

Everything up until now is saved in an imageschema, ready to be committed to the database.

```
    var saveImage = new Image({
      id : newId,
      title : req.query.title,
      publisher : req.query.publisher,
      views : 0,
      comments : [],
      date : Date.now()
    });
```

A request is then sent to the image site. When a response is given, the content-type is checked for 'image/jpeg'. If this criteria is met, handlerImageDownload is called upon with the response, path and a callback. This writes the image to disk. The size is checked and sent back. If the image is larger than what is defined, the image schema is committed to the database. If the content-type is not what is expected, we respond with Node.js writehead HTTP status code 400. If the server could not save the schema or a general error, it will respond with 500.

```
    request.get(req.query.image).on('response', function(
        response) {
      var contype = response.headers['content-type'];
      // Checking if we got an image back
      if (!contype || contype.indexOf('image/jpeg') !== 0) {
        res.writeHead(400);
        res.end('RESPONSE NOT AN IMAGE');
      } else if (contype === 'image/jpeg') {
        handlerImageDownload(response, imagepath, function(size)
            {
          // Verifying the datasize
          if (size >= 10000) {
            // Save Mongoose schema to database with error
                handling
            imageSchema.save(function (err, saveImage) {
              if (err) {
```

```
              res.writeHead(500);
              res.end('ERROR WRITING TO DB');
            }
          });
        } else {
          res.writeHead(500);
          res.end('NOT ENOUGH DATA FOR IMAGE');
        }
      });
      // Everything went ok!
      res.writeHead(200);
      res.end('SAVED IMAGE, WROTE TO DATABASE');
    } else { // Everything else is responded with internal
       server error
      res.writeHead(500)
      res.end('INTERNAL SERVER ERROR');
    }
  });
 });
});
```

The last routing handler is used for adding a new comment to an already existing image. This is basically using the Mongoose findOne function and getting the image ID that is provided in the URL query. Then the comment text which is also part of the query, is pushed into the array of comments that each image has. If the push is successful, it will respond with a JSON object.

```
/* Get new comment */
router.get('/newComment', function(req, res, next) {
    var comment = { name: req.query.name, text: req.query.text,
       date: Date.now() };
      Image.findOneAndUpdate({'id' : req.query.id},
        {$push: {comments: comment}},
        {$safe: true, /*upsert: true*/},
          function(err, imageComment) {
            res.jsonp(imageComment);
          }
      );
});
```

Express uses a template engine to respond to the request. This is done by using res.render. At runtime the engine uses a static file and replaces variables in a template file with actual values [20].

The following code is an example of this. Here the static file 'index' is used to map variables to the one in the template. The left column is the variable names in the file and the right column is variables in the previous code snippets. The code as a whole can be read in appendix A.1.3.

```
    res.render('index', {
        title : 'flackr',
        images : images,
        lastID : images[0].id,
        topView : req.topImage
    });
```

This is the jade template index file. It can be used to extend the layout template for the whole page, and everything in this file is an extension of this. The variable title is in the layout file and not seen here. Rest of the variables are from the previous code snippet.

```
extends layout

block content
  p Number of entries in database: #{lastID}
  if topView
    p The image with most views: #{topView.id}
  // For each object in objectarray
  each image in images
    #grid
      .grid-element
        a(href='#{image.id}')
          img(src='#{image.id}.jpg')
```

3.3.2 Architecture

Node.js and Express.js

Node.js and Express go hand in hand. It is already mentioned that an express-generator is used to get started with a basic setup. However, it is required to install other packages first.

```
sudo apt-get update
sudo apt-get install npm nodejs-legacy
```

Node.js and npm has now been installed. npm can now be used to install express. First of all a directory is created for the application. Then express and the express-generator is installed. The last module is used to create a simple basic folder and files structure, which is explains in detail at a later point. To generate a basic starting point for the application, the command express "application name" is executed. Since the directory folder already exists, express will prompt a message saying that the destination is not empty, this can be ignored.

```
mkdir flackr
cd flackr
sudo npm install express --save
cd ~
sudo npm install express-generator -g
express flackr
```

Running the express "application name" command, gives the following output.

```
destination is not empty, continue? [y/N] y

   create : flackr
   create : flackr/package.json
   create : flackr/app.js
   create : flackr/public
   create : flackr/public/javascripts
   create : flackr/public/images
   create : flackr/public/stylesheets
   create : flackr/public/stylesheets/style.css
   create : flackr/routes
   create : flackr/routes/index.js
   create : flackr/routes/users.js
   create : flackr/views
   create : flackr/views/index.jade
   create : flackr/views/layout.jade
   create : flackr/views/error.jade
   create : flackr/bin
   create : flackr/bin/www

   install dependencies:
     $ cd flackr && npm install

   run the app:
     $ DEBUG=flackr:* npm start
```

As displayed above it sets up the basic directories and files required to run a "hello world" application. It generates the file package.json which is basically metadata for the application, it contains all the dependencies required. This means that whenever we install a package with npm, it will save the module name and its version to that file. Running the command npm install in the application directory will check whether all the dependencies listed in package.json are installed.

MongoDB

MongoDB provides packages for 64-bit long-term-support Ubuntu releases.[21] Which means that it can be installed with apt on Ubuntu 14.04 LTS. We have followed the guide that is written in the docs for MongoDB[21]. First of all the official MongoDB public GPG Key is immported, as shown in step 1. Then in step 2 a list file is created. Reload the local package database. And finally, install the latest stable version og MongoDB.

```
1. sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --
   recv EA312927

2. echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-
   org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/
   mongodb-org-3.2.list

3. sudo apt-get update

4. sudo apt-get install -y mongodb-org
```

Now that MongoDB is installed, we can start configuring it. First of all it is checked which port and which IP addresses it listens to. The configuration file is located at /etc/mongod.conf. The port is 27017 by default, and the IP address is 127.0.0.1 also known as localhost. For the simplicity it is changed it to 0.0.0.0 for now. As shown in the config file extract below. The complete file is found in the appendix MongoDB Configuration FileE.

```
# network interfaces
net:
  port: 27017
  bindIp: 0.0.0.0
```

Linux uses a known memory management system called THP, it reduces the overhead of TLB lookups on machines with large amounts of memory by using larger memory pages. This is known to be a performance issue with MongoDB, considering that database workloads often perform poorly with THP, because they tend to have sparse rather than contiguous memory access patterns.[22] MongoDB have provided a script that can be implemented to disable THP. The script, located in appendix D, has the following functionality: It locates the path of two files, depending on which Linux distro, then it concatenates 'never' into these configuration files. It must then be configured to start on system boot, and after a restart THP will be disabled.

```
case $1 in
  start)
    if [ -d /sys/kernel/mm/transparent_hugepage ]; then
      thp_path=/sys/kernel/mm/transparent_hugepage
    elif [ -d /sys/kernel/mm/redhat_transparent_hugepage ]; then
      thp_path=/sys/kernel/mm/redhat_transparent_hugepage
    else
      return 0
    fi

    echo 'never' > ${thp_path}/enabled
    echo 'never' > ${thp_path}/defrag

    unset thp_path
    ;;
esac

sudo chmod 755 /etc/init.d/disable-transparent-hugepages
sudo update-rc.d disable-transparent-hugepages defaults
```

After MongoDB has been configured, the service will need to be started again.

```
sudo service mongod start
```

MongoDB is now running as a service and should also be listening for incoming traffic as specified in the configuration file earlier. To check if its configured correctly, it is possible to verify that the mongod service listens to the correct IP address.

```
sudo netstat -tulpen | grep 27017

tcp    0    0   0.0.0.0:27017    0.0.0.0:*   LISTEN    106   9517
    1318/mongod
```

Furthermore, users and authentication has to be created and enabled respectively. Creating user accounts in MongoDB is a step by step procedure, which has to be followed strictly and in chronological order. First of all the shell is entered as shown in step 1. Secondly, the admin database is created for storing users. Step 3 creates a user with dbOwner access, meaning it is able to administrate users and its databases. Of course the password should be set to something a little more complex, however, for readability it is set to something simple. The user for the Flackr database is also created, which is to be used with a mongoose connection, as explained later on in this chapter. This user only has a read/write role.

```
1.  mongo

2. use admin
```

```
3. db.createUser(
     { user: "userAdmin",
       pwd: "userAdminPassword",
       roles: [{role: "dbOwner", db: "admin" }]
     }
   )

4. db.createUser(
     { user: "flackr",
       pwd: "flackrPassword",
       roles: [{role: "readWrite", db: "flackr" }]
     }
   )
```

The mongo shell is now exited, and the configuration file for MongoDB is opened again. The commenting sign from the security section is removed and the following is added.

```
security:
  authorization: enabled
```

The service is then restarted and we can log in with the admin user, to make sure everything is correct. It is required to specify in which database the user credentials are located to be able to log in.

```
sudo service mongod restart

mongo -u "userAdmin" -p "userAdminPassword" --
    authenticationDatabase "admin"
```

To verify that the users exist, the following commands are executed and the output given shows the result.

```
>use admin
> db.getUsers()
[
        {
                "_id" : "admin.userAdmin",
                "user" : "userAdmin",
                "db" : "admin",
                "roles" : [
                        {
                                "role" : "userAdminAnyDatabase",
                                "db" : "admin"
                        }
                ]
        },
        {
                "_id" : "admin.flackr",
                "user" : "flackr",
                "db" : "admin",
                "roles" : [
                        {
                                "role" : "readWrite",
                                "db" : "flackr"
                        }
                ]
        }
]
```

Now that MongoDB has been installed and the users have been created, it is time to look at replication between two MongoDB instances. After launching a new server MongoDB is installed the same way as above. Once again the configuration file is edited, to include the replica set name.

```
replication:
  replSetName: rs0
```

Then one of the servers were elected to be the primary or master server. In order to get started with replication, there are some commands that have to be issued from the master, these have to be entered in the mongo shell.

```
rs.initiate()   # Start replication set
rs.conf()       # Configure replica set
rs.add('ip add of slave')
```

If MongoDB successfully added a slave, it returns an "ok" message.

```
Return: {"ok":1}
```

To verify if the replication is correctly configured, the status command can be issued.

```
rs.status()
```

```
"_id":0,"stateStr":"PRIMARY" and "_id":1,"stateStr":"SECONDARY"
```

which means that its all synchronized.

MongoDB is now ready for use with the application.

Mongoose

Mongoose is used to communicate with the database in a simple manner. It needs to be installed using the npm package manager.

```
npm install mongoose
```

PM2

PM2 needs to be installed with npm.

```
sudo npm install pm2 -g
```

This command installs PM2 globally on the application server. When launching your app for the first time, PM2 takes a number of commands. Some of them are optional. The following command starts the the application by running the file app.js, which requires that you are in the directory of the application files. Otherwise, the path must be specified. To keep things simple, in case one has more than one application, it is wise to give it a name, using the –name "name" option.

By default PM2 logs to the folder /.pm2/logs/, and to have more specific log entries it is possible to add a timestamp to the log entry.

```
pm2 start app.js --name "flackr" --log-date-format="YYYY-MM-DD
    HH:mm Z"
```

PM2 has a number of useful commands for handling your application. "pm2 list" will list all the applications that are running with relevant information such as memory used, PID, status and uptime. The command "pm2 monit" initiates a live monitoring session of the running applications, showing the amount of memory currently being used. The rest are self-explanatory.

```
pm2 list
pm2 monit
pm2 restart "appname"
pm2 stop "appname"
```

Now that pm2 is installed, it is useful to have it launch at system boot. To be able to do this the "pm2 startup" command is used.

```
pm2 startup ubuntu
[PM2] You have to run this command as root. Execute the
   following command:
    sudo su -c "env PATH=$PATH:/usr/bin pm2 startup ubuntu -u
       ubuntu --hp /home/ubuntu"
```

Nginx

Nginx is fairly simple to install and configure in this setup. It can be downloaded and installed using APT. To enable caching in Nginx, a directory has to be defined in the configuration file.

```
1. sudo apt-get update
   sudo apt-get install nginx

2. sudo mkdir -p /data/nginx/cache
   sudo chown <username> /data/nginx/cache
   sudo chmod 700 /data/nginx/cache
```

Now that Nginx is installed the configuration file located at /etc/nginx/sites-available/default has to be edited. The levels parameter specifies how the cache will be organized. Nginx will create a cache key by hashing the value of a key. The levels configured below will dictate that a single character directory with a two character subdirectory is created. These characters are taken from a hashed key, configured in the next line with proxy_cache_key. The keys_zone parameter defines the name for the current cache zone, with parameters for how much metadata to store. Each megabyte is able to hold approximately 8000 entries of keys. The max_size parameter is the maximum cache size.[23]

```
 proxy_cache_path /data/nginx/cache/ levels=1:2 keys_zone=
    backcache:8m max_size=50m;
proxy_temp_path /data/nginx/cache/tmp;
proxy_cache_key "
   $scheme$request_method$host$request_uri$is_args$args";
```

The parameter proxy_cache_valid can be specified multiple times , it allows us to configure how long to cache certain values depending on the status code. In this case the hits are stored for 5 seconds and the cache for 404 not found expire every minute.

```
proxy_cache_valid 200 302 5s;
proxy_cache_valid 404 1m;
```

Since Nginx is used to load balance between a pool of servers, the server IP addresses need to be defined in the configuration. The port Nginx will listen to is specified.

```
upstream flackrapp {
  server "application server ip"
  server "second application server ip"
}

server {
  listen 80;
  listen [::]:80;
```

Furthermore, it is required to specify when Nginx has to use the cache. This is done by saying in the locations section where it is proxied to a backend. Nginx shall use the backcache zone that was configured earlier. It is then configured that it needs to contain an indicator as to whether the service requests a fresh, non-cached version, with the proxy_cache_bypass directive and its parameter. An extra header is added, named X-Proxy-Cache. This sets a header that allows it to be seen if the request resulted in a cache hit, miss or bypassed. If no caching is found, the request is passed to a server from the upstream pool.

```
  location = / {
    proxy_cache backcache;
    proxy_cache_bypass $http_cache_control;
    add_header X-Proxy-Cache $upstream_cache_status;

    proxy_pass http://flackrapp;
   }
  location / {
    proxy_pass http://flackrapp;
  }

}
```

Nginx is now installed and configured properly for use with the Flackr application. It might take a few minutes for it to correctly pass on the connections. The configuration file can also be found in appendix F.

GlusterFS

The setup of GlusterFS requires two servers, each with one Cinder volume attached. These can be created in OpenStack and will not be explained here. In addition, naturally, GlusterFS client software has to be installed on the application servers.

On all servers that are using GlusterFS in any way it is required to update and download some repositories. Software-properties-common is installed to easily handle PPA's with APT [24]. It is also recommended to install attr, in order to access the extended attributes added by GlusterFS, but this is not necessary.

```
sudo apt-get update
sudo apt-get install software-properties-common
sudo apt-get install attr
```

The community GlusterFS PPA is added.

```
sudo add-apt-repository ppa:gluster/glusterfs-3.5
sudo apt-get update
```

GlusterFS is now ready to be installed. The following commands should only be executed on the servers intended to be used as GlusterFS-Servers.

```
sudo apt-get install glusterfs-server
```

Now that GlusterFS server is installed on both nodes, the process of setting up storage volumes can begin. From one of the hosts it is necessary to peer the other, the gluster01 server was chosen for simplicity as a master[25].

```
sudo gluster peer probe ip-address-of-gluster02
```

A filesystem has to be created on the attached disks.

```
mkfs.xfs -i size=512 /dev/disk/by-id/virtio-b6dde619-feed-455b-a
```

A Directory is created, where disks are mounted. Note that the directory has a different name on server 2.

```
sudo mkdir -p /data/glusterbrick01

sudo mount /dev/disk/by-id/virtio-b6dde619-feed-455b-a /data/
    glusterbrick01
```

Now that the disks have been mounted to the storage directories, replication can set up with GlusterFS[26].

```
sudo gluster volume create imagestorage replica 2 transport tcp
    ip-address-of-gluster01:/data/glusterbrick01 ip-add-of-
    gluster02:/data/glusterbrick02 force
```

As instructed by GlusterFS the volume has to be launched.

```
sudo gluster volume start imagestorage

volume start: imagestorage: success
```

GlusterFS is installed on the clients, which are the web application servers.

```
sudo apt-get install glusterfs-client
```

After installing the client software, a directory has to be created and then it is possible to mount the remote volume.

```
sudo mkdir /data/imagestorage

sudo mount -t glusterfs ip-add-of-gluster01:imagestorage /data/
    imagestorage/
```

It can also be mounted automatically at system boot.

```
sudo vi /etc/fstab

ip-add-of-gluster01:/imagestorage /data/imagestorage glusterfs
    defaults,nobootwait,_netdev,backupvolfile-server=ip-add-of-
    gluster02,direct-io-mode=disable 0 0
```

All that is left now is to change the ownership on the web application servers, so that node can access the directories and files.

```
sudo chown ubuntu:ubuntu /data/imagestorage
```

Now the files can be copied from the application folder to the network filesystem. In the following example, the files are copied from the source /images and to destination /data/imagestorage/ created previously.

```
cp ~/images /data/imagestorage
```

Now a redundant storage pool with GlusterFS is set up for usage with our application and the variable config.image.folder can be specified with the imagestorage directiory to redundantly save the images.

3.4 Traffic generation

3.4.1 Introduction

An important part of our project is to create a script that is able to generate traffic to the application, in order to simulate real environment traffic. Currently our employer uses his script by spawning several virtual machines that executes his script. The script in his case adds new users or comments on a post to his current site. The traffic script developed in this architecture should have similar functionality and parameters, in order to easily use it in his current setup. The script has to be able to add a new picture, comment to it, visit the frontpage and visit all the pictures individually.

3.4.2 Implementation

The script has several possible command line arguments. One of the command line arguments has to be the IP address for the entry point of the application. It is checked whether the option "U" is included in the beginning of the script, more of the options are used later on in the script, but as the option "U" is vital to the script, it is checked first. As seen below, the accepted arguments are "U,P,V,F,C,H,R" where options U, V, and C take a parameter. If option 'U' is found, the parameter is set to be the URL variable to be used throughout the script. If it can't find the option, then an error is thrown, as the script would be useless without the URL.

```
FOUND_OPTION_U = False
OPTIONS, REM = getopt.getopt(sys.argv[1:], 'U:PV:FC:HR')
for alt, argum in OPTIONS: #Check if arguments have -U
    if alt in '-U':
        URL = argum
        FOUND_OPTION_U = True

    if not FOUND_OPTION_U:
        print "NO URL GIVEN"
        sys.exit()
```

The option "F" is a simple function for visiting the front page of our application. It works by creating a folder with a random name. A simple wget is performed to download all of the contents of the front page, including its pictures, CSS and HTML code. The wget will simulate someone visiting the front page. After it has done the wget, it deletes all the content downloaded.

```
def front_page():
    """ Function, visit frontpage of flackr"""
    rand = get_int(0, 1000) # Get random int for folder name

    folder = TEMPDIR + str(rand) # make folder path of tempdir
       and random int
    os.mkdir(folder) # Make new folder
    os.chdir(folder) # Change directory to folder
    os.system("wget -t 2 -T 5 -p -q "+URL)
      # Wget all elements from webpage inc css,logo,photos
    os.system("rm -r "+folder)  # Remove folder and its contents

    print "visited front page"
```

Before creating the function for adding a new picture, functions to generate random names and titles has to be implemented. There is a website located at https://randomuser.me/api/ which generates a JSON object with lots of random information as shown in the appendix I. The python "urllib2" module, which is an extensible library for opening URLs[27], is used to download the JSON object. The object is then parsed into a variable, using a python JSON parser. However, as only the first and last names are required, they can be extracted from the parsed JSON object, and returned as a combined name.

```
def get_random_name():
    """ Function for getting a random user name"""
                        # Downloads a json document with random
                             user information
    response = urllib2.urlopen('http://api.randomuser.me/1.0/?
        nat=gb,us&inc=name&noinfo').read()
    parsed_json = json.loads(response)  # Parse the json into
        variable
    firstname = parsed_json["results"][0]["name"]["first"].title
        ()
         # Select firstname and capitalize
    lastname = parsed_json["results"][0]["name"]["last"].title()
         # select Lastname and capitalize

    name = firstname + " " + lastname      # Add names together

    return name      # Return generated name
```

The scripts also has a function for creating picture titles. It is almost exactly the same as the get_random_name function. The only difference is shown in the extract below. Where the last name value from the JSON object is extracted, and a "The" is added to make it sound like a title. This is a simple implementation, but it works.

```
temp = parsed_json["results"][0]["name"]["last"].title()
      # Select only lastname and capitalize
title = "The" + " " + temp    # Make it sound like a title

return title
```

Now that the name and title functions have been completed, the function for adding new pictures to the website can be constructed. It gets the name and title variables from calling each individual function. The python "urllib" module is used to access the URL with the route "/insert", to add a new user. For the pictures a placeholder image site at http://lorempixel.com is used, where it is possible to access the URL with parameters for the image size, and it gives a random picture from their database in return. If the HTTP code is anything but 200 (OK), the database should not insert the data.

```
def new_picture():
    """ Function for adding new picture to website"""
    name = get_random_name()  # Get name and title
    title = get_random_title()
          # uses urllib module for opening up connection to
            webpage
          #  and add a new picture using the /insert route
    response = urllib.urlopen(URL+"insert/newUser?title=" + \
                              title + "&publisher=" + name \
                             + "&image=http://lorempixel.com
                                 /1280/720/")
    #print "%s" % response
    print "added new picture"
```

Considering that there is a function for visiting the front page, it makes sense to have a function that could perform the action of visiting each individual image site. It is based on the same idea as for visiting the front page, however, the function takes a parameter for whether to visit all the sites, or visiting a certain amount of random site. In the beginning of this project, we were provided with a file containing random sentences from our employer, which he currently uses for his Bookface application. This is implemented in the function for generating comments. Before the function is able generate comments, there needs to be a function for extracting a sentence from the file. It works by opening the file, jumping to a random line, checking if the sentence is acceptable and of correct length and then returns it.

```
def get_random_text():
    """ Get a random sentence from the sentences file"""
    find_line = False  # bool variable

    if os.path.isfile(FILEPATH):  # check if file can be found
        temp = open(FILEPATH)     # Open file
        text = temp.readlines()   # read all the lines
        while find_line is False:  # Until it finds a proper
            line
             line = get_int(0, LINE_LIMIT)  # Get random line
             if is_sentence(text[line]) is False:
               # Use is_sentence function to check if acceptable
                 find_line is True    # Set bool value to true to
                     stop while loop
                 sentence = check_sentence_length(text[line])
                 # Check if sentence is too long
                 return sentence   # Return the sentence
        temp.close()
    else:
        print "Could not find file!!!"
```

The function for generating comments also takes a parameter, it can either comment to all images once or comment a certain amount to random sites. If the function is called with parameter R, for random, it will generate half times the amount of currently existing pictures. However, it will comment to a random picture each time, unlike the all command, which comments to all pictures from 1 to the end.

```
def gen_comment(option):
——omitted comment——
    items = get_webpage_items()
    opt_random = False
    if option == 'R':
        item_range = items / 2
        opt_random = True
    else:
        item_range = items
    for item in range(1, item_range + 1):
     # comment to all or random, +1 is compensation for range.
        name = get_random_name()     # Get random name from
            function
        text = get_random_text()     # Get a sentence from the
            sentences file
        if opt_random:      # If option R, get random int for ID
            item_id = get_int(1, items)
        else:
            item_id = item  # Open up the url /insert route
        response = urllib.urlopen(URL+"insert/newComment?id=" \
                                 + str(item_id) +"&name=" +
                                    name \
                                 + "&text=" + text)
        #print "%s" % response
        print "generated comment to: " + str(item_id)
```

One of the command line parameters that can be used is the option -R. This is a simple implementation of a random function, meaning that when the option -R is passed it will either generate a new picture, comment randomly, view random sites or just visit the front page. Being a simple implementation, it generates a random integer between 0 and 100, based on the value it will call a specific function.

```
def random_operation():
    """ Function for doing one of the provided options
      randomly """
    random_number = get_int(0, 100)  # Gets int in provided
        range
    if random_number <= 25: # Depending on int it does one of
        the options
        new_picture()
    elif random_number > 25 and random_number <= 50:
        gen_comment('R')
    elif random_number > 50 and random_number <= 75:
        gen_views('R')
    elif random_number > 75 and random_number <= 100:
        front_page()
```

The last part of the script is the main. Which calls the individual functions based on the command line options and their arguments.

```
def main():
    """ Main menu, takes option as command line argument"""
    for opt, arg in OPTIONS:  # And based on input, calls
        required functions
        if opt in '-P':
            new_picture()
        elif opt in '-F':
            front_page()
        elif opt in '-C':
            if arg == 'A' or arg == 'R' or arg == 'O':
                gen_comment(arg)
            else:
                raise ValueError("wrong ARGUMENT! A, R or O!!")
        elif opt in '-V':
            if arg == 'A' or arg == 'R':
                gen_views(arg)
            else:
                raise ValueError("Wrong argument! A or R!!!")
        elif opt in '-H':  # For now it adds H_RANGE users
            for _ in range(H_RANGE):
                new_picture()

            front_page()  # Visits frontpage
            gen_comment('A')  # comments to all
            gen_views('A')    # views all
        elif opt in '-R':
            random_operation()

if __name__ == '__main__':
    main()
```

3.5 MEAN Stack Security

In recent years full stack development methods like the MEAN stack has become a very popular way to develop applications and systems. This new way of development brings along some security problems and complications that we did not encounter before. In the past it was normal to have dedicated system administrators for the different parts of the system. Nowadays when dealing with full stack development the developers have a lot more responsibility when it comes to security and testing compared to before.

3.5.1 MongoDB

MongoDB is a NoSQL database that is well known for being JavaScript friendly, and can be largely used in the same manner as the well known MySQL database. But there is a common misconception that MongoDB is not vulnerable to SQL injection-type attacks. Although MongoDB is not vulnerable to the SQL language attacks, it is still vulnerable to various injection attacks and cross-site request forgery (CSRF) threats. To improve the MongoDB security, we have implemeted some security related key points and tips from the official MongoDB documentation[28], which helped us create a secure MongoDB environment.

- **Enable Access Control and Enforce Authentication**
  Authentication requires that all clients and servers provice valid credentials before they can connect to the system. This can be enabled in the MongoDB config.
- **Restrict connections to the database**
  Make sure the database is not accessible from the public Internet and restrict which other entities are allowed to connect to the MongoDB server. It is recommended to only allow your application/web servers access to the MongoDB server. In this project it is possible to restrict connections to the database by configuring the OpenStack configurations or use the 'iptables' application on the database server.

- **Restrict MongoDB to listen only on relevant interfaces**
  By default MongoDB will bind to all available network interfaces on the server. To limit this to only the relevant network interfaces, it is possible to add "bind_ip = IP_ADDRESS" in the MongoDB config.
- **Enable TLS/SSL for all incoming and outgoing connections**

3.5.2 Express.js

Express.js is a minimal and flexible server-side Node.js web application framework. Express.js is vulnerable to various injection and cross-site attacks and is also exposed to all of Node.js's underlying vulnerabilities. To improve the Express.js security, we have implemeted some security related key points and tips from the official Express.js documentation[29], which helped us create a secure application environment.

- **Use the latest version**
  Since the release of Express version 4, older versions(2.x and 3.x) are no longed maintained. This means that security and performance issues in these versions will not be fixed.
- **Disable HTTP headers**
  There are several different HTTP headers that is vulnerable to exploits, and it is therefore recommended to disable some of them. It is possible to turn the headers off manually, but the easiest way is to install Helmet. Helmet can help protect the application from well-known web vulnerabilities by setting HTTP header appropriately. [30]
- **Avoid known web vulnerabilities**
  Just like any other web application, Express.js applications can be vulnerable to a variety of well known web attacks. It is therefore important to keep up with general web vulnerabilities to ensure a secure web application.
- **Use Transport Layer Security(TLS)**
  It is strongly recommended to enable TLS encyption in Express.js, especially if the application deals or transmits sensitive data. Since Flackr is meant to be deployed in an closed environment, TLS is considered to not be important.

3.5.3 Angular.js

Angular.js is a JavaScript framework developed and maintained by Google. Since the beginning of Angular.js, the framework have had some issues that it could be vulnerable to various cross-site scripting attacks. These vulnerabilities have been patched and fixed in the latest versions of Angular.js. Since Angular.js is a part of Google, it is very well maintained security wise, and therefore the best security advice is to keep it up to date. [31]

3.5.4 Node.js

Node.js enables the building of web applications with extensive server-side and networking capabilities, and enables real-time two-way communications between the client and server. Arguably the defining component of the MEAN stack, Node.js is not without its own vulnerabilities—not only does it inherit all JavaScript-related vulnerabilities, but also gains some new attack vectors while executing on the server side. As we did with MongoDB and Express.js, we have implemeted some security related key points and tips from the official Node.js documentation[32], which helped us create a secure application environment.

- **Avoid running applications as root**
  Running the Node.js application as root can lead to the whole system going down in case of an error/bug in the app.
- **Use an HTTP server/proxy to forward requests**
  It is strongly recommended that an HTTP server/proxy handles incoming requests for the application. In the first version of Flackr, the application does NOT use an HTTP server/proxy and is therefore very vulnerable to various attacks. The second version of Flackr uses an Nginx reverse proxy to handle all requests between the application and the external network.
- **Be careful placing sensitive data**
  When deploying front end applications make sure not to expose sensitive information in the source code, as it can be readable by anyone.
- **Static code analysis**
  It is always important to test your code for errors and problems. In our project we have used JSLint to analyze our JS code.

3.6 Testing

Testing is an essential part of this project, especially since Flackr is a system which is required and expected to work in a real environment. By testing the architecture and application, it ensures that what we have created does what it is supposed to do, and we were able to discover parts where improvements could be made. Since an agile development method is used in this project, testing has been involved throughout the entire development period.

Testing of Flackr have mainly involved five different testing methods:

- System testing
- Static code analysis
- Performance testing
- User manual review
- Traffic generation scrip testing

3.6.1 System testing

By doing System testing, we evaluate the complete system's compliance with its specified requirements. This made it possible for us to find errors, fix them and implement overall improvements to the architecture.

3.6.2 Static code analysis

For all of our JavaScript code we have tested it using JSLint, which is a code quality tool that looks for problems in JavaScript programs. If JSLint detects a problem, it will return a message describing the problem and where the problem is located within the code. For our traffic generation script we have used pylint, which has basically exactly the same functions as JSLint, but for Python code.

3.6.3 Performance testing

Except to just test our Nginx server by performing overall system tests, we also ran some performance tests to evaluate our Nginx load balancer. There are several performance tools meant for this kind of testing, and one of them is ApacheBench. ApacheBench is a benchmarking tool for measuring the performance of HTTP web servers. The usage of a benchmarking tool provided us with information on how well our Nginx load balancer interact with our web servers.

3.6.4 User manual review

Since the students will follow the user manuals when setting up this architecture, it is very important that the user manuals provide the correct information. Therefore we have completed the setup process of Flackr multiple times using the user manuals.

3.6.5 Traffic generation script testing

Testing of our Traffic generation script included both Static code analysis of the script itself, but also basic system testing. In this case system testing involves pushing a lot of traffic onto the servers by running the script in a loop, as well as testing all the functionality of the script.

4 Second Architecture - Factory

The second architecture is developed using AMQP, a message layer middleware. This system is supposed to represent a factory which has an assembly line. The factory has an entry point where incoming jobs will be accepted and placed on a queue. In the factory there are multiple workers which take jobs from the queue and start processing them. When a job has successfully gone through the whole assembly line it is completed and will be stored in a database. By developing an architecture using message queueing, we wanted to present the potential of asynchronous messaging, and how applications easily can connect to each other as components of a larger application. Operational challenges like scalability, availability, uptime and single points of failure are also covered. Although the architecture in this case represents a factory and an assembly line as the base idea, the architecture can represent anything that requires processing through several steps.

4.1 Technology

Just like in the first architecture this section starts off by going through all the modules, applications and other technologies that are required to make a complete message queueing system. The idea of this section is to get a basic understanding of how the different technologies work. Specific description on how the technologies are used in the project will be presented in Design, which is the next section.

4.1.1 RabbitMQ

RabbitMQ is an open source message broker software that implements the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover.

4.1.2 Flask

Flask works as a microframework that aims to have a simple but extensible core. You decide on what database, form validation or anything else as long as there is a supported extension for it[33].

4.1.3 Pika

Pika is a pure-Python implementation of the AMQP 0-9-1 protocol that tries to stay fairly independent of the underlying network support library.[34]

4.1.4 PM2

PM2 is a production process manager for Node.js applications with a built-in load balancer. It allows you to keep applications alive forever, to reload them without downtime and to facilitate common system admin tasks.

4.1.5 MongoDB

MongoDB is the leading open source NoSQL database, which uses a document-oriented data model.

4.1.6 Munin with RabbitMQ plugin

Munin is a free and open-source computer system monitoring, network monitoring and infrastructure monitoring software application. It offers monitoring and alerting services for servers, switches, applications and services. In addition to the standard munin setup, it uses the RabbitMQ munin plugin to monitor the RabbitMQ message brokers.

4.1.7 RabbitMQ Management Plugin

The rabbitmq-management plugin provides an HTTP-based API for management and monitoring of a RabbitMQ server, along with a browser-based UI and a built-in CLI tool.

4.2 Design

This architecture will not have any versions, which was decided during a meeting with our employer. An e-mail with verification can be found in appendix H.

4.2.1 Application

Entrypoint

The point of entry in this architecture is a server running a Python Flask application, API and RabbitMQ. The role of this server is to accept jobs in a certain format, add relevant data of its own, and push it on the local RabbitMQ queue ready for workers to take over.

Workers and queues

The worker machines will be running a Python application, it will be the same application on every worker, but it has differences in each individual configuration file. The application will have functions for consuming messages from a given queue, process the job and add its own time stamp to it and eventually pass it on to the next queue in the assembly line.

Safe

The safe server is considered to be the end of the assembly line and is responsible for completing the jobs that pass through the factory, and write the jobs to the database. The idea is that it listens to jobs from its local queue, picks one, processes it and puts a time stamp on it before it goes on and writes it to the database. Furthermore, it also runs a Python Flask application that displays all the jobs on a website, which is discussed in the next paragraph.

Finished Job Entry System

All jobs that have successfully gone through the Factory system will be placed in a database. A web application will fetch all jobs from this database, and present it on a web page. All jobs will consist of 6 different variables:

- ID: Each job have it's own unique ID
- Name: Name of the user that requested the job
- Jobtype: Represents what kind of job it is, can vary from 1-5. Each jobtype causes different processing.
- Start Time: The timestamp when the job was initially started.
- Stop Time: The timestamp when the job was completed.
- Worker Stamps: Array of stamps. One for each worker that it passes through.

This web application will help the students monitor the RabbitMQ architecture by keeping track of completed jobs that were sent through the whole system.

Common

All the machines described above uses PM2 to handle their applications. PM2 also provides basic monitoring as well as logging for each individual application. The Python applications that are involved with either consuming or producing messages uses Pika, which is the recommended Python client by the RabbitMQ team[35].

4.2.2 Architecture

As shown in figure 5, the architecture resembles a basic assembly line. Initially, there will be only one worker taking jobs off the queue, processing them and adding a time stamp before pushing it onto the next queue. Eventually, it will start to pile up on the queues because of the massive quantity of incoming jobs. This can be solved by adding new worker servers and scale vertically which also solves the concern for single point of failures. For example, if worker01 goes down for maintenance, worker03 can continue and keep everything operational. Entrypoint01, queue01, safe and mongodb01 are all single point of failures here. But the main idea is to discuss important operational topics, this is intentional and by design. In a real world production environment, this should not be the case.

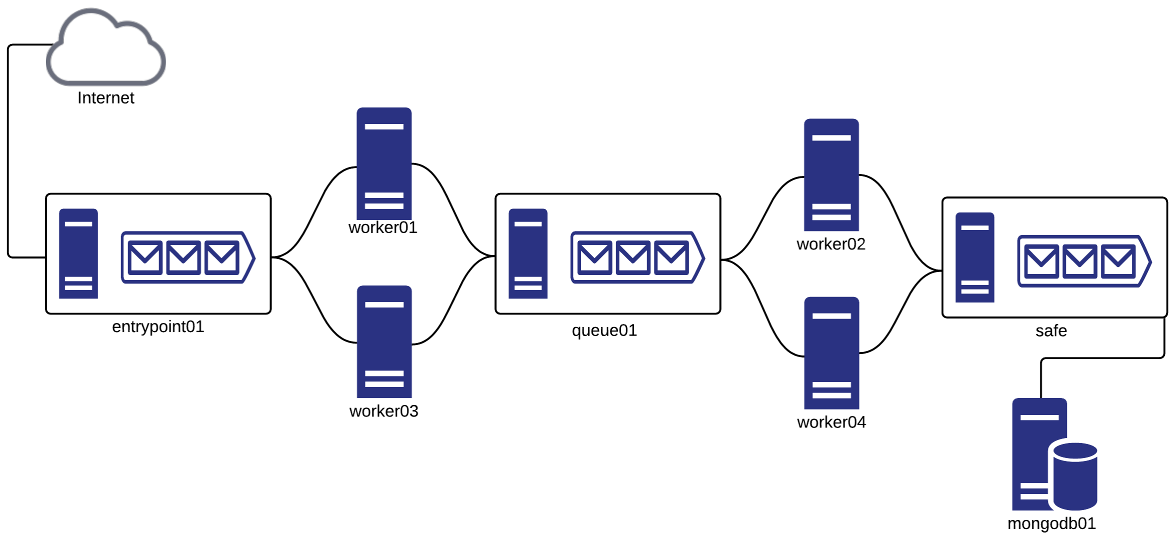


Figure 5: Deployment view of Factory

4.2.3 Monitoring

Management and monitoring something that is immediately interesting when planning this architecture. The idea is to create a simple and effective way for the students to get an overview of their system. There are several possibilities when it comes to RabbitMQ Management and Monitoring.[36]. The most well known alternatives would be Nagios and Zabbix, but they are both way too heavy and advanced for our project, so therefore we decided to go with the following applications.

Munin with RabbitMQ Plugin

The primary monitoring tool for the Factory will be Munin with a RabbitMQ plugin. Munin will by default provide alot of information about a server/node, but it is also possible to provide additional information by easily installing third party plugins. By using the RabbitMQ Munin Plugin, munin will provide the students and the course supervisor with an overview of all the RabbitMQ servers, and individual information about every server. All munin information will automatically be presented in graphs.

RabbitMQ Management Plugin

In addition to Munin, the usage of the official RabbitMQ Management Plugin is a great way to manage and monitor our RabbitMQ servers/nodes. This plugin will provide us with a browser-based GUI with queue management and monitoring for each server/node. The downside with this plugin is that it can only manage a single server/node, while in Munin it is possible to monitor multiple nodes in the same GUI.

4.3 Implementation

4.3.1 Application

All Python applications in this architecture use a YAML based configuration file. Which contains environment variables such as IP addresses, ports, RabbitMQ credentials and the queue names.

Entrypoint

As mentioned earlier, this server is running a Flask Flask application with the Flask-RESTful extension. In order to simulate users sending in jobs, an exposed REST API is provided, in the same way someone from a shop is sending in an order to a factory that gets processed on an assembly line. The application itself is simple, it accepts jobs only if the request method is POST, "Content-Type" header is "application/json", and it has a name. If the criteria is met, it will add a starting time and initialize an empty array. When this is done, Pika will take over. Since the Entrypoint is also running RabbitMQ, Pika will set up the connection to itself and push the job onto the queue waiting for workers to take over.

The RESTful API is built as resources on top of pluggable views. An example of a resource can be seen in the follwing code where thhe class works as a resource. Functions are typical HTTP methods exposed from the resource.

```
class NewJob(Resource):
    def post(self):
———————Omitted code———————
```

The next code snippet is an endpoint. The resource is added to the API and matched to an url.

```
api.add_resource(NewJob, '/job')
```

The code as a whole can be seen in appendic A.2.4.

Workers

The workers are assigned with a specific task, consume and produce messages from and to RabbitMQ. The application code can be viewed in full detail in the appendixA.2.1.First of all, the application needs to declare a connection to RabbitMQ. Since it has to both consume and produce a message, it has to have a connection with two different brokers.

The connection uses the credentials saved in the configuration file, which are the credentials created when installing RabbitMQ. It then declares a connection to the queue by using the "parametersConsume" variable. This variable uses the IP address found in the configuration file. In case the server is unreachable, it tries to establish a connection 5 times with a 15 second delay in between. It is the same case for "parametersProduce", which has the queue it needs to deliver its messages to.

```
credentials = pika.PlainCredentials(cfg['rabbitUser'], cfg['
    rabbitPsw'])
parametersConsume = pika.ConnectionParameters(cfg['receiveFrom
    '], cfg['port'], \
                                    '/', credentials, retry_delay
                                        =15, \
                                    connection_attempts=5)
parametersProduce = pika.ConnectionParameters(cfg['produceTo'],
    cfg['port'], \
                                    '/', credentials, retry_delay
                                        =15, \
                                    connection_attempts=5)
```

Now that the variables have been declared, a connection can be established between the workers and queues. Considering that neither of the workers will be at the entrypoint, it is natural for it to always be listening to a queue for jobs, before it can do work and pass it along to the next queue. Hence, the consuming connection is started first.

```
connectionConsume = pika.BlockingConnection(parametersConsume)
channelConsume = connectionConsume.channel()
```

Before it is possible to start consuming messages, the queue has to be declared. It is declared as durable so that RabbitMQ does not lose the items in a queue when rebooting.

```
channelConsume.queue_declare(queue=cfg['queue'], durable=True)
```

When consuming messages, a callback function has to be used in order for the application to subscribe to a queue. Whenever a message is placed on the queue, the callback function is called by the Pika library. The function loads the message as a JSON object, it then reads the sleep tag, which in this scenario is a basic "jopType" representing the type of job and how long it is going to process it. Processing in this case is just a simple sleep, it serves as a method of doing tentative work. Furthermore, a stamp is added to the JSON object. Before it passes the message along to another function, a basic acknowledgement is sent to RabbitMQ saying that it has consumed the message and that RabbitMQ is free to delete it from the queue. This ensures that no jobs are lost, in case of a connection failure. The message can now be passed along to the producer function.

```
def callback(ch, method, properties, body):
    """ Function for consuming messages from message queue"""
    print " [*] " + timeStamp() + " Accepted job"
    data = json.loads(body)
    sleep_tag = int(data["jobType"])
    time.sleep(sleep_tag)
    stamptime = timeStamp()
    data["stamp"].append({cfg['worker']:stamptime})
    channelConsume.basic_ack(delivery_tag=method.delivery_tag)
    print " [*] " + timeStamp() + " Done consuming, starting
        producing"
    produce(data)
    print " [*] " + timeStamp() + " Waiting for next job"
```

The producer function establishes a connection to the RabbitMQ server, where it declares the queue, and publishes the message that was sent as a parameter from the callback function.

```
def produce(item):
    """ Function for producing a message to a message queue"""
    #Start producing connection
    connectionProduce = pika.BlockingConnection(
        parametersProduce)
    channelProduce = connectionProduce.channel()
    channelProduce.queue_declare(queue=cfg['queue'], durable=
        True)
    message = json.dumps(item)
    #Publish message to queue
    channelProduce.basic_publish(exchange='', routing_key=cfg['
        queue'], \
                                  body=message, \
                                  properties=pika.BasicProperties(
                                      \
                                      delivery_mode=2, \
                                      content_type='application/
                                          json'))
    connectionProduce.close()
    print " [*] " + timeStamp() + " Done producing"
```

Earlier it is mentioned that to be able to consume messages a callback function is required. Pika takes messages from the queue and passes them to the callback function.

```
channelConsume.basic_consume(callback, queue=cfg['queue'])
channelConsume.start_consuming()
```

Safe

As shown in our diagram earlier, the safe machine is the end of the assembly line, and is responsible for wrapping up the jobs and writing them to the database. The last workers in the assembly line, send their messages to the queue at the safe machine. The task of the safe machine is to listen to its local queue, process the jobs one at a time and write them to the database. The application starts off by connecting to the database which is critical, because if there is no connection, then no jobs will be written to the database. The python syntax "try and except" is implemented to catch the error if no connection could be established. Using the native error handling from the pymongo package, if there is no established connection to the database the application shuts down.

```
#Connect to database
try:
    dbCon = MongoClient('mongodb://' +cfg['databaseIP'] + \
         ':' + cfg['databasePort'] + '/')
    dbCon.server_info()
    db = dbCon[cfg['db']]
    posts = db[cfg['collection']]
    collection_exist = db.collection_names()
except pymongo.errors.ServerSelectionTimeoutError as err:
    print " [!]  Could not connect to DB: %s" % err
    print " [!]  No items will be written to DB!!"
    sys.exit()
```

Considering that the application has to connect with RabbitMQ, Pika is once again required, but since the code for connecting is very similar it is not discussed here.

Nevertheless, there is still a callback function that will be called when messages appear on the queue. Some of the code is the same as on a worker, however, instead of adding a timestamp to the array, it sets the "timeStop", which is the time when the job is considered to be finished.

The most important task for the safe application is to deal with the database. It is responsible for setting the "ID" of a job, as well as writing the job iteself to the database. This is implemented by quering the database when establishing a connection, and checking if the collection used exists. If it doesn't exist then the "ID" is automatically set to 1, because MongoDB doesn't create any collection before something is written to it. This will only happen the first time the application accepts a job.

The Pymongo syntax "find()" is used to query the database. When using "find()", it always returns it as a pymongo cursor object, which has to be iterated in order to extract values. To safely try and extract the "ID" value, "try and except" is implemented to catch the python KeyError if the "ID" field doesn't exist. This would indicate an error in the database and the application is stopped.

If the "ID" has been set, data can be inserted with the syntax "posts.insert_one(data)", unless there is a connection failure, then the job is successfully written to the database.

```
def callback(ch, method, properties, body):
    """ Function for consuming messages from queue"""
    print " [*] " + time_stamp() + " Accepted job"
    data = json.loads(body)
    time.sleep(2)
    data["timeStop"] = time_stamp()
    if not collection_exist:
        data["id"] = 1
    else:
        try:
            get = posts.find({}, {"id":1, "_id":0})\
                    .sort([("_id", pymongo.DESCENDING)]).limit
                        (1)
            for doc in get:
                try:
                    val = int(doc['id']) +1
                    data["id"] = val
                except KeyError:
                    print " [!] " + time_stamp() + \
                        "Error in database! ID field not found"
                    sys.exit()
        except pymongo.errors.ConnectionFailure as err:
            print " [!] " + time_stamp() + \
                    " Could not get ID from database!! %s" % err
            sys.exit()
    try:
        posts.insert_one(data)
        print(" [*] Wrote to database, ID: {}".format(data['id
            ']))
    except pymongo.errors.ConnectionFailure as err:
        print " [!] " + time_stamp() + \
                " Could not post to database! %s" % err
        sys.exit()
    channel.basic_ack(delivery_tag=method.delivery_tag)
```

As previously stated, a method to call the function is needed, when items appear on the queue.

```
channel.basic_consume(callback, queue=cfg['queue'])
channel.start_consuming()
```

This concludes the consumption of messages from RabbitMQ. Furthermore, the safe machine also runs one of the Flask applications that presents all the jobs on a straightforward website, which will be described in the next paragraph.

Finished Job Entry System

This is a simple Flask application using PyMongo for queries and the built-in jinja2 template engine, with a specific purpose to show a certain number of entries that have gone through the architecture with stamps and an end date.

```
@app.route('/')
def index():
    ''' POST GET '''
    # Query database for entries
    query = collection.find().sort("id", -1).limit(cfg['
        frontpageLimit'])
    return render_template('index.html', entries=query)
```

The application is written using the normal route decorator to bind a function to URL. In this case it binds the index page to the function named index. When the query is completed, it is sorted by the last ID and the amount of entries is limited by a setting in the configuration file.

Flask has built-in error handling for HTTP status codes, however, a custom made error page for code 404 is implemente,d which displays a simple 404 with an image of a cute kitten. This is done by using abort to end the request early and the errorhandler function will then handle the exception.

```
@app.errorhandler(404)
def page_not_found(error):
    ''' Error handling for 404 '''
    return render_template('404.html', err=error), 404
```

The variables are then passed to the jinja engine, which will populate the web page. The website consist of a index.html template, which is used to build up a HTML page when Flask gets a request. The template is located in appendix A.2.3.

4.3.2 Architecture

RabbitMQ

RabbitMQ has to be installed on all servers that deal with messaging and queuing, which includes all servers except the database. The repository has to be added to the sources list, and the RabbitMQ public key has to be added to the trusted key list.

```
echo 'deb http://www.rabbitmq.com/debian/ testing main' |
    sudo tee /etc/apt/sources.list.d/rabbitmq.list

wget -O- https://www.rabbitmq.com/rabbitmq-signing-key-public.
    asc | sudo apt-key add -
```

Before installing the rabbitmq-server package it is recommended to perform an update.

```
sudo apt-get update
sudo apt-get install rabbitmq-server
```

When RabbitMQ is installed it is required to add a user with adminstrator rights. This is the username and password that is used by Pika in our workers to authenticate to RabbitMQ.

```
sudo rabbitmqctl add_user factory factoryPass
sudo rabbitmqctl set_user_tags factory administrator
sudo rabbitmqctl set_permissions factory ".*" ".*" ".*"

sudo service rabbitmq-server restart
```

Common

All of the workers require a couple of packages installed to be able to download and execute the application code, located in a git repository. Additionally, it is required to install a Python package for Python to be able to communicate with RabbitMQ.

```
sudo apt-get install -y python python-pip git

sudo pip install pika
```

Now that the tools for executing the application code is installed, software for handling the applications has to be installed. Considering that PM2 is able to handle Python applications as well, it fits this architecture. Installing PM2 requires that npm and Node.js is already installed.

```
sudo apt-get update
sudo apt-get install -y npm nodejs-legacy

sudo npm install pm2@latest -g
```

MongoDB

As MongoDB is the chosen database for this architecture, it requires configuration, however, it is exactly the same configuration as covered in the first architecture. Hence, it is not discussed here again. The complete commands for installation will be covered in the user manuals.

4.3.3 Monitoring

Monitoring was something our employer requested to be implemented into this architecture. It was therefore necessary to create a simple and effective way for the students to get an overview of their system. The Factory architecture will consist mainly of two different monitoring applications.

Munin with RabbitMQ Plugin

Munin is available in two different software packages depending on which role the server as assigned.

Munin Master:
The Munin master is responsible for gathering data from Munin nodes. For Munin to work it is required to have a working Munin master server. Most of the Munin configuration and setup process is implemented on the Munin master. Considering that the course supervisor will host and manage the Munin master, it is only required that the students configure and set up their Munin nodes.
Munin Node:
The Munin node is the agent process running on the servers that shall be monitored by the Munin master. When Munin is installed, it will automatically start on boot and listen on port 4949/TCP accepting connections from the Munin master. The process of setting up a Munin node and getting it up and running is really simple.

The Munin node should be installed on appropriate servers that are going to be monitored.

```
sudo apt-get update
sudo apt-get install -y munin-node
```

When the Munin node software package is successfully installed, the connection between the Munin node and the Munin master needs to be established. To create this connection, it is required to allow the IP address of the Munin master in the Munin node config, located at /etc/munin/munin-node.conf.

```
allow ^IP_OF_MASTER_NODE$
```

If the Munin Server is correctly configured, it should now be possible to monitor the Munin Node through the Munin master GUI. Munin will by default present a lot of basic system information about the server. Although this information is useful, the whole idea by using Munin in this architecture is to monitor RabbitMQ. To enable monitoring for RabbitMQ it is required to install a third-party plugin.

```
cd /usr/share/munin/plugins
sudo git clone https://github.com/ask/rabbitmq-munin.git
sudo cp rabbitmq-munin/* .
```

After downloading the plugin, symlinks between the plugins and the Munin folder have to be created. To see which plugins that require a symlink, it is possible to run the munin-node-configure script.

```
sudo munin-node-configure --shell

sudo ln -s '/usr/share/munin/plugins/rabbitmq_connections' '/etc
    /munin/plugins/rabbitmq_connections'
sudo ln -s '/usr/share/munin/plugins/rabbitmq_consumers' '/etc/
    munin/plugins/rabbitmq_consumers'
sudo ln -s '/usr/share/munin/plugins/rabbitmq_messages' '/etc/
    munin/plugins/rabbitmq_messages'
sudo ln -s '/usr/share/munin/plugins/
    rabbitmq_messages_unacknowledged' '/etc/munin/plugins/
    rabbitmq_messages_unacknowledged'
sudo ln -s '/usr/share/munin/plugins/
    rabbitmq_messages_uncommitted' '/etc/munin/plugins/
    rabbitmq_messages_uncommitted'
sudo ln -s '/usr/share/munin/plugins/rabbitmq_queue_memory' '/
    etc/munin/plugins/rabbitmq_queue_memory'
```

The munin-node-configure script is also able to check if plugins are enabled and symlinked. If all the rabbit_* plugins states are yes|yes it means that the plugins are enabled.

```
sudo munin-node-configure --suggest
```

After symlinking the plugins, it is recommended to restart the node service.

```
sudo service munin-node restart
```

The final step of the plugin installation is to add the plugins in the /etc/munin/plugin-conf.d/munin-node configuration file, and give them root permissions.

```
[rabbitmq_connections]
user root

[rabbitmq_consumers]
user root

[rabbitmq_messages]
user root

[rabbitmq_messages_unacknowledged]
user root

[rabbitmq_messages_uncommitted]
user root

[rabbitmq_queue_memory]
user root
```

RabbitMQ Management Plugin

The management plugin is included in the RabbitMQ distribution, and therefore the only thing that requires for it work, is to enable it and have a user with administrator permissions.

```
rabbitmq-plugins enable rabbitmq_management
```

Since the 3.3.0 release of RabbitMQ, the server will prevent access using the default guest/guest credentials except via localhost. Hence, to access the RabbitMQ database remotely it is required to have a user with admin rights. This user was created earlier when installing RabbitMQ. The Management plugin browser-based UI is now available by logging in with the username and password at http://ip-address-of-server:15672/.

4.4 Traffic generation

4.4.1 Introduction

A part of our requirements is to implement a script that is able to generate traffic in order to simulate a real environment. As mentioned earlier, this architecture is something that will be set up in a short period of time at the end of the course, to illustrate certain operational challenges.

For this script we wanted a way to use the API provided by the application entrypoint, to add data that can be processed by workers in the assembly line. The previous script has been used as a base and modified to fit our requirements for this architecture. Furthermore, the script has two new argument options which will be explained in depth in the following section.

4.4.2 Implementation

With the script from the first architecture as a baseline, we kept the base structure with arguments using "getopt" for parsing the command line arguments. Functions named ¨front_page¨ and ¨get_random_name¨ will be used as well. The two new functions are option 'O' for one new entry and "M" for many new entries. Everything is still written in Python, and uses a mix of standard modules and an additional module named 'request'. This module will do the actual HTTP request towards the entrypoint to simulate a person sending a new job entry.

A single new entry has several steps. First it use the get_random_name function which is the name of the person who sent in a job order. Second step is to generate a random integer that will simulate what type of job it is. The third step is to build up the request, where we define what the payload and headers will consists of. Eventually the request is executed with simple error handling.

```
def new_entry():
    """ Function to add a single new entry"""
    name = get_random_name()
    jobType = randint(1, 5)
    # Constructing the request
    payload = {'name' : name, 'jobType' : jobType}
    headers = {'Content-Type': 'application/json'}
    # Execute the request with error handling for general and
       404
    try:
        req = requests.post(URL, data=json.dumps(payload),
           headers=headers)
    except requests.exceptions.RequestException as err:
        print_error_msg(err)
        sys.exit(TRACEBACK)
    else:
        if req.status_code == 404:
            print_error_msg(req.raise_for_status())
            sys.exit(TRACEBACK)
    # Print out the reply text
    print req.text
    return req.text
```

Many new entries make use of the previous function. However, it loops a certain number of times based on the number and adds a new entry each time.

```
def many_new_entries(number):
    """ Function to add several new jobs, using previous
       functions"""
    # For number in range, add a new entry
    if number.isdigit():
        for i in range(int(number)):
            new_entry()
    else:
        print_error_msg("Passed variable is not a number.
           Exiting...")
        sys.exit(TRACEBACK)
    print "Done! Sent %s new entries." % number
```

4.5 RabbitMQ Security

This section will discuss security hardening approaches for RabbitMQ, which uses the AMQP protocol. Security is an important part of application layer protocols, and therefore it is very relevant for us to clarify what we have done to improve RabbitMQ security in this architecture.

4.5.1 Access Control

Access Control contains authentication and authorisation. Authentication in RabbitMQ is defined as "identifying who the user is", and authorization is defined as "determining what the user is and is not allowed to do".[37]

When a new RabbitMQ server is started up, it initialises a database with default login credentials, which is "guest" as username and password. It is strongly recommended to create a new user, and delete the default "guest" user. Since the 3.3.0 release of RabbitMQ, the server will prevent access using the default guest/guest credentials except via localhost. So to access the RabbitMQ database remotely it is required to create a new user with admin rights through the server command-line interface.

4.5.2 Messaging transport security

AMQP based solution support transport-level security using TLS. It is highly recommend enabling transport-level cryptography for a message queue. Since RabbitMQ 3.4.0, TLS/SSL is automatically disabled to prevent the so called POODLE attack.[38] Using the TLS/SSL support in RabbitMQ will provide protection of the communication between the client and server. To implement TLS/SSL in RabbitMQ, it is required to do some configuration on the RabbitMQ server. Since this architecture is running in a closed environment, we have decided to not enable TLS on our RabbitMQ servers. More information on how to enable TLS in RabbitMQ can be found here.

4.5.3 RabbitMQ Security Checklist

Additionally we have created this security checklist to improve the security in the RabbitMQ environment:

- Make sure the RabbitMQ client is up to date - earlier versions of RabbitMQ had multiple cross-site scripting(XSS) vulnerabilities.
- Enable TLS/SSL
- RabbitMQ was not meant to be exposed directly to the public Internet, so if this is a requirement, it is recommended to use for example HAproxy in front of the RabbitMQ servers.
- Make sure the default login credentials for the RabbitMQ database is either removed or changed.

4.6 Testing

Testing is an essential part of our project, especially since our Message Queueing system is required and expected to work in a real environment. By testing the architecture and system, we ensure that what we have created does what it is supposed to do, and we are able discover parts where improvements can be made. Since we are using an agile development method, testing has been involved during the entire development period.

For the Factory we have mainly tested the the architecture in five different ways:

- System testing
- Static code analysis
- Performance testing
- User manual review
- Traffic generation scrip testing

4.6.1 System testing

By doing System testing, we evaluate the complete system's compliance with its specified requirements. This made it possible for us to find errors, fix them and implement overall improvements to the architecture.

4.6.2 Static code analysis

To analyze the code in our traffic generation script for the Factory, we have used pylint, which is a source code bug and quality checker for the Python programming language.

4.6.3 Performance testing

Except to just test the Factory by performing overall system tests, some benchmarking tests have also been launched to evaluate the performance of RabbitMQ. To perform this kind of testing on the RabbitMQ servers, we used PerfTest. PerfTest is a performance testing tool.It starts up zero or more producers and consumers, and reports the rate at which messages are sent and received, along with the latency (i.e. time taken for messages to pass through the broker). The usage of a performance testing tool provided us with information on how well our RabbitMQ servers perform, and where improvements should be implemented.

4.6.4 User manual review

Since the students will follow the user manuals when setting up this architecture, it is very important that the user manuals provide the correct information. Therefore we have completed the setup process of the Factory multiple times using the user manuals.

4.6.5 Traffic generation script testing

Testing of our traffic generation script included both Static code analysis of the script itself, but also basic system testing. In this case system testing involves pushing a lot of traffic onto the servers by running the script in a loop, as well as testing all of the functionality in the script.

5 Discussion

5.1 Results

In this chapter we will discuss and reflect on what we have accomplished throughout this bachelor thesis.

5.1.1 Project outcome

We have created two architectures to be used in IMT3441 database and application administration. This included deciding upon a design, investigating which technologies to use, developing an application that suits the architecture, developing a tool for generating traffic, and writing documentation.

Our first architecture is Flackr, a web application that shares images on a website. Built using the MEAN stack, PM2, GlusterFS and Nginx. Here the students will be challenged with installing, maintaining and upgrading the application. As they upgrade the application to newer versions, they can expect overall improvements regarding performance, stability and security.

The second architecture is named Factory, which uses technologies such as RabbitMQ, MongoDB, Flask, Pika and RESTful API to create a workflow where jobs pass through an assembly line. Once the jobs are done, they are committed to a database. Students will get a theoretical and practical view on operational issues such as single point of failure, bottlenecks, scalability and high uptime. Additionally, the students are introduced to message queues which is widely used in system architectures.

The learning aspect has been broad, interesting and challenging. Looking into different technologies, learning how they work by installing and trying them out, and figuring out what could work together was definitely an interesting experience. Deciding whether to develop our own application or finding an existing alternative wasn't an easy decision for us. Neither of us has experience developing web applications. We expected to write code in form of scripts or simple applications. Once we realised that most of the web applications we found, was either changing too often, used as a non-functional example, or were not really relevant. We took on the challenge of developing on our own, and saw it as an opportunity to learn something new.

5.1.2 What did we not do

Pretty much all of our project goals were met, both learning and performance wise. When it comes to the requirements specification regarding functionality, usability, reliability and security from chapter 2, these are all fulfilled. There is still room for improvement, and this will be discussed in a later section.

5.1.3 What could have been done differently

User testing is something that could have been done earlier, and in a much larger scale. We could have been more consistent when it comes to writing documentation during the different design and installation phases.

5.1.4 Technologies

In order to find relevant technologies, we spent a great amount of time researching alternatives. Criteria such as what works well together, other peoples experiences with it, and the documentation played an important role in deciding.

For Flackr, there were not that many full stack alternatives to LAMP. The task description mentioned both Node.js and NoSQL, so the decision to use MEAN was an easy one. GlusterFS was mainly chosen because it is already teached in the course, it works well as a distributed network file system, and could handle the load required transferring the images between the application servers. Because of this it was chosen as the preferred alternative instead of Ceph. We briefly discussed using object storage after having a split-brain issue with GlusterFS, and we would have looked more in to it if the current version of SkyHiGh supported it. Nginx and PM2 were chosen when we were looking into running a Node.js application in production. We did not chose to look for other alternatives as both documentation and experiences was overall positive. Using Python as the scripting language for the trafficscript was decided early. We wanted to learn more about the language as it is rising in popularity, and it is able to perform wanted functionality.

Factory is based around the idea of using a message queuing system. We tried out both RabbitMQ and ZeroMQ, after careful consideration based on the requirements stated earlier, we ended up going with RabbitMQ. Overall, it was easier to understand and the documentation was better. However, if we had more time to study the documentation and test, then ZeroMQ would probably be the preferred choice. The reasoning behind MongoDB was simple, we already had a positive experience with implementing it with Flackr and it seemed like a good idea to save time. This way we could spend more time on other areas. In order to send in data to the architecture, we started looking into RESTful API. The group had experience using Flask. Alternatives were Node.js, but the desire to learn more Python made the choice simple. Initially, we did not plan on showing the entries written to the database, however, after a meeting with the employer who requested to implement this for educational purposes. At this point the development with Flask using API had started, and therefore the decision was made to continue using Flask for this as well.

5.1.5 Time usage

In our initial planning phase we created a Gantt chart to illustrate our project schedule. As we were tasked with designing two architectures, we felt that it was natural to divide our time into two equal segments. Additionally we set aside time at the end of each segment to write documentation. Dividing our time into segments was a bit of a risk, due to the fact that it provides little flexibility. On the other hand, it gave us a date to work towards during the project, where an architecture would have to be completed in order to progress and move on to the next. This worked well for us, we were able to follow our initial planning scheme and made great progress throughout the project period. Using a Gantt chart in our planning phase really payed off, as it provided us with a time line to follow. The Gantt can be seen in appendix L.

5.1.6 Complications

Wrong JSON format

In the traffic generating scripts, we used a website named randomuser.me. Which is a free, open-source API for generating random user data. However, after using it for a while, we had problems when parsing the JSON data trying to extract the information that we wanted. After much troubleshooting, we noticed that randomuser.me had changed the structure of their JSON, when upgrading to a new version of their website. According to their documentation, it is possible to specify which version of the API to use, thus we implemented this in our function to ensure that the traffic scripts would work as intended, despite any future updates.

Lorempixel

Flackr is a website where displaying images is the main feature. We needed a website to extract images from. After some googling around we ended up using Lorempixel, which was stable and had the exact features that we wanted. Initially, we had a great success until it went down in March, around the time we were starting the documentation phase. At this point, our error checking wasn't the greatest and we learned a lot about error handling when it comes to using status code and content type. All of these things mentioned are now checked for whenever an insert is requested. If status code is 200, content type in the HTTP header is 'image/jpeg', and the file size is larger than 10 kilobytes then the schema is committed to the database. So in the end, something good came outfrom the initial issue.

MongoDB Authentication

We had some issues when attempting to implement authentication with MongoDB. According to MongoDB documentation, there are two ways to implement authentication. First of all we started the Mongo shell, then added users with the db.CreateUser command, which was not the issue. The issue arose when trying to restart the Mongo shell with parameters for adding authentication, after trying this we were not able to start the Mongo shell again. The reason for this, is that earlier we used a configuration file for specifying which port and which IP addresses are allowed, and when trying to enable authentication by starting the Mongo shell with the –auth option, which did not work. It seems as if MongoDB prefers that one either uses a configuration file or only using parameters for starting the Mongo shell. After starting from scratch, we only used the configuration file for enabling access control and did a sudo service mongod restart instead, then we were able to implement authentication without any new issues.

5.2 Group evaluation

Group organization is described in the appendix. The group leader role have not been used as much since the group dynamic was great. Most of the discussions have ended in an agreement, and when not, they usually occured because the message wasn't conveyed clearly enough. The only exception to this is a discussion about how much time we wanted to spend developing the application once we decided to do it ourselves.

By using tools like Toggl to track time spent on a day to day basis, Trello to organize tasks, Git to document code, and Google Docs to document everything else we have had a good flow throughout the project. Everything we wanted to do has been a card in Trello and discussed in our internal group meetings. This is something that have worked exceptionally well for us.

We have met with our supervisor on a weekly basis to discuss various topics. Having someone to bounce off ideas and topics have been helpful.

5.3 Further Development

In this section we discuss how the architectures and application can be further improved and developed.

Trafficscript independence from Lorempixel

Relying on a third party site for images it not necessarily a bad idea, until it either completely shuts down or maintenance needs to be done. Controlling this part by generating an image on the workers running the user script, then transferring it over would be a way to control this aspect and ensure it is working at all times.

Implement more operational difficulties

During the installation, configuration and operation of these architectures, the student will become familiar with operational challenges like scaling, reliability, uptime, and single point of failures. Adding ways to present monitoring and centralized logging are other examples that should be implemented. These are important topics in the real world, and by exposing the students could potentially better prepare them.

6 Conclusion

After spending 5 months working on this project, we feel that we have been able to apply the knowledge gained throughout the three years here at NTNU into our bachelor thesis work.

At first, we were sceptical to the task description, but after the first meeting with the employer, we quickly understood that the choice of taking on this bachelor project was the correct decision. Our experiences with IMT3441 as a course has been very positive, as we consider it as one of the best during the time here. Being able to help further broaden the knowledge of future students by developing alternative architectures has definitely been a motivating factor.

When we started off this project we set ourselves a few goals, which we hoped to accomplish during the project period. One of the goals was to understand and differentiate between different service architectures, in order to do this we set of with designing two fully independent architectures, named Flackr and Factory. When designing these architectures, a lot of time was spent installing, configuring and implementing different technologies. Additionally, we used our fundamental education to write scripts, use databases, develop applications and implement security by design. The only learning goal we feel we have spent less time with is networking, instead the focus was more on further enhancing previously mentioned skills. Another goal was to research and understand best practices, which helps us prepare for working with real-world IT operations.

We have also learned a lot when it comes to working with larger projects in a group. This is something that we believe we will benefit from in a real working environment.

Last not but least, we are satisfied with what we have accomplished in this bachelor thesis, and hope that our employer can use this in his course to improve the educational outcome.

Bibliography

[1] Wikipedia, "iterative and incremental development". https://en.wikipedia.org/w/index.php?title=Iterative_and_incremental_development&oldid=680280418. (Online; Visited 20 February 2016).

[2] Google, "python style guide". https://google.github.io/styleguide/pyguide.html. (Online; Visited 27 March 2016).

[3] Owasp, "web application top 10 risks 2013". https://www.owasp.org/index.php/Top_10_2013-Top_10. (Online; Visited 23 April 2016).

[4] Owasp, wapplication security guide for cisosw. https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf. (Online; Visited 23 April 2016).

[5] Cnet.com, "newbie's guide to flickr". http://www.cnet.com/news/newbies-guide-to-flickr/. (Online; Visted 1 March. 2016).

[6] Mean.io, "mean". http://mean.io. (Online; Visited 12 Feb. 2016).

[7] Nginx, "using nginx as http load balancer". http://nginx.org/en/docs/http/load_balancing.html. (Online; Visited 15 Feb. 2016).

[8] Facebook, "hammering usernames". https://www.facebook.com/notes/facebook-engineering/hammering-usernames/96390263919/. (Online; Visited 28 March 2016).

[9] Changelog, "dark launching software features". http://changelog.ca/log/2012/07/19/dark_launching_software_features. (Online; Visited 28 March 2016).

[10] Express, "index page". http://expressjs.com/. (Online; Visited 5 March 2016).

[11] Express faq, "route listings". http://expressjs.com/en/starter/faq.html. (Online; Visted 1 Feb. 2016).

[12] Express, "router api". http://expressjs.com/en/4x/api.html#router. (Online; Visited 5 February 2016).

[13] Amazon, "aws reference architecture". https://aws.amazon.com/architecture/. (Online; Visited 23 February 2016).

[14] Nokola breznjak, "using nginx as a reverse proxy in front of your node.js application". http://www.nikola-breznjak.com/blog/nodejs/using-nginx-as-a-reverse-proxy-in-front-of-your-node-js-application/. (Online; Visited 15 Feb. 2016).

[15] Howtoforge, "why you should always use nginx with microcaching". https://www.howtoforge.com/why-you-should-always-use-nginx-with-microcaching. (Online; Visited 15 Feb. 2016).

[16] Imgur, "tech tuesday: Our technology stack". https://blog.imgur.com/2013/06/04/tech-tuesday-our-technology-stack/. (Online; Visited 15 Feb. 2016).

[17] Mongodb, "mongodb and mysql compared". https://www.mongodb.com/compare/mongodb-mysql. (Online; Visited 10 May 2016).

[18] Openstack, "attach a single volume to multiple hosts". https://specs.openstack.org/openstack/cinder-specs/specs/kilo/multi-attach-volume.html. (Online; Visited 4 March 2016).

[19] Stackoverflow, "how to store node.js configuration settings". http://stackoverflow.com/questions/5869216/how-to-store-node-js-deployment-settings-configuration-files. (Online; Visited 25 Feb 2016).

[20] Express, "using template engines with express". http://expressjs.com/en/guide/using-template-engines.html. (Online; Visited 25 February 2016).

[21] Mongodb, "mongodb installation guide". https://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/. (Online; Visited 12 Feb. 2016).

[22] Mongodb documentation, "transparent huge pages". https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/. (Online; Visited 15 Feb. 2016).

[23] Digialocean, "understanding nginx http proxying, load balancing, buffering, and caching". https://www.digitalocean.com/community/tutorials/understanding-nginx-http-proxying-load-balancing-buffering-and-caching. (Online; Visted 1 March. 2016).

[24] Getting started install glusterfs. http://www.gluster.org/community/documentation/index.php/Getting_started_install. (Online; Visited 13 March 2016).

[25] Gluster, "getting started configure glusterfs". http://www.gluster.org/community/documentation/index.php/Getting_started_configure. (Online; Visited 13 March 2016).

[26] Glusterfs, "how to create a redundant storage pool using glusterfs on ubuntu servers". https://www.digitalocean.com/community/tutorials/how-to-create-a-redundant-storage-pool-using-glusterfs-on-ubuntu-servers. (Online; Visited 9 March 2016).

[27] Python docs, "urllib2 - extensible library for opening urls". https://docs.python.org/2/library/urllib2.html. (Online; Visited 14 March 2016).

[28] Mongodb, "security guide". https://docs.mongodb.org/manual/MongoDB-security-guide-master.pdf. (Visited 24 April 2016).

[29] Express production best practices: Security. http://expressjs.com/en/advanced/best-practice-security.html. (Online; Visited 24 March 2016).

[30] npmjs, "help secure express/connect apps with various http headers". https://www.npmjs.com/package/helmet. (Online; Visited 26 March 2016).

[31] Angularjs, "angularjs security". https://docs.angularjs.org/guide/security. (Online; Visited 25 April 2016).

[32] Risingstack, "node.jssecurity tips". https://blog.risingstack.com/node-js-security-tips/. (Online; Visited 24 April 2016).

[33] Flask, "foreword". http://flask.pocoo.org/docs/0.10/foreword/. (Online; Visited 15 April 2016).

[34] Pika documentation, "introduction to pika". http://pika.readthedocs.io/en/latest/index.html#. (Online; Visited 3 April 2016).

[35] Rabbitmq, "python - get started". https://www.rabbitmq.com/tutorials/tutorial-one-python.html. (Online; Visited 20 March 2016).

[36] Rabbitmq management and monitoring. https://www.rabbitmq.com/how.html#management. (Online; Visited 1 May 2016).

[37] Rabbitmq, "rabbitmq authentication". https://www.rabbitmq.com/access-control.html. (Online; Visited 24 March 2016).

[38] Us-cert, "ssl 3.0 protocol vulnerability and poodle attack". https://www.us-cert.gov/ncas/alerts/TA14-290A. (Online; Visited 23 April 2016).

A Application code and configuration files

A.1 Flackr

A.1.1 app.js

```
/*   Flackr
     IMT3912 Bacheloroppgave IMT
*/

// Mongoose module, configuration file and schema
var mongoose = require('mongoose');
var config = require('./config');
var Schema = mongoose.Schema;

// Database connection
connection = ('mongodb://' + (config.mongodb.username) + ':' + (
    config.mongodb.password) + '@' + (config.mongodb.ip) + '/' +
     (config.mongodb.name));
console.log(connection);

var options = {
  auth: {authdb: "admin"}
};

mongoose.connect(connection, options , function(err) {
  if (err) {
    console.log('Error connecting to database', err);
  } else {
    console.log('Connected to database');
  }
});

// Make a scheme based on template
var Image = require('./imageSchema.js');


// Requires from generator
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

// Define express instance
var app = express();

// Application endpoints
var routes = require('./routes/index');
var insert = require('./routes/insert');


// View engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// Uses
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico'))
    ); not used
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
app.use(express.static(config.image.folder ));
app.disable('x-powered-by');

// npm start
app.listen(3000, function(err) {
  console.error('press CTRL+C to exit');
});

// Loading router modules in the application
app.use('/', routes);
app.use('/insert', insert);


// Catching 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// Development error handler
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

// Production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});


module.exports = app;
```

A.1.2 routes/index.js

```
var express   = require('express');
var fs        = require("fs");
var path      = require("path");
var config    = require ('../config');
var mongoose  = require('mongoose');
var Image     = require('../imageSchema.js');
var router    = express.Router();


/* GET picture */
router.get('/', function(req, res, next) {
  // If feature flag is enabled, find image with the most views
  if (config.image.topviews == 1) {
    Image.findOne({}).sort({views: -1}).exec(function(err, topIm
        age) {
      if (err) {
        console.log('Error was thrown...');
        throw (err);
      } else {
        console.log("Most views ID: " + topImage.id);
        req.topImage = topImage;
      }
    });
  }
  else {
    req.topImage = null;
  }

  // Find top based on frontpagelimit and render it
  Image.find({}).sort({date: -1}).limit(config.image.frontpageli
      mit).exec(function(err, images) {
    if (err || !images.length) {
      res.status(404);
      res.render('error', {
        title : 'flackr',
        message : 'Object returned from database is empty.'
      });
    }
    else {
      res.render('index', {
        title : 'flackr',
        images : images,
        lastID : images[0].id,
        topView : req.topImage
      });
    }
  });
});

/* GET :id */
router.get('/:id', function(req, res, next) {
  // Find one image based on route
  Image.findOne({id: req.params.id}, function(err, oneImage) {
    // Checks if findOne returns err, nothing or id is invalid
    if(err || oneImage == null || isNaN(req.params.id) == true)
        {
      // Return 404 with error page
      res.status(404);
      res.render('error', {
        title : 'flackr',
        message : 'Document not found.'
      });
    } else {
      // Increment if page is visited
      oneImage.views++;
      oneImage.save();

      res.render('image', {
        title : 'flackr',
        image : oneImage
      });
    }
  });
});

module.exports = router;
```

A.1.3 routes/insert.js

```
var express = require('express');
var mongoose = require('mongoose');
var Image = require('../imageSchema.js');
var Image = mongoose.model('Image');
var request = require('request');
var fs = require('fs');
var config = require ('../config');

var router = express.Router();

var placeholder = 'http://placehold.it/1280x720';

// Downloads the image we got, splitted out so this can be done
    async
function handlerImageDownload(response, localPath, callback) {
  // Creates a stream to the path provided
  var writeStream = fs.createWriteStream(localPath);
  // Writing data
  response.pipe(writeStream);
  // When done writing data, check the size and send it back
  response.on('end', function() {
    fs.stat(localPath, function (err, stats) {
      size = stats['size'];
      callback(size);
    });
  });
};

// Query the database to find the ID for the new insert
function handlerDatabaseQuery(callback) {
  // Mongoose query
  Image.findOne().sort({'id': -1}).exec(function(err, query) {
    // If error from database, return with error
    if(err) {
      console.log('Error occured');
      return;
    } else if (query === null) {
    // Special case; if database is empty we don't know the last insert ID
      var newId = 1;
      console.log('New ID: ' + newId);
      callback(newId);
    } else {
      // Error handling and special case, last is that we get an ID
      console.log('Last ID: ' + query.id);
      var newId = query.id+1;
      console.log('New ID: ' + newId);
      callback(newId)
    }
  });
}

/* GET New User  */
router.get('/newUser', function(req, res, next) {
  // Database handling
  // Query to find the last ID
    handlerDatabaseQuery(function(newId) {

    // Filepath concat
    var imagepath = config.image.folder + newId + '.jpg';

    // Getting the scheme ready from query string and ID
    var imageSchema = new Image({
      id : newId,
      title : req.query.title,
      publisher : req.query.publisher,
      views : 0,
      comments : [],
      date : Date.now()
    });

    // Does a request for an image to the URL specified
    request.get(req.query.image).on('response', function(response) {
      var contype = response.headers['content-type'];
      // Checking if we got an image back
      if (!contype || contype.indexOf('image/jpeg') !== 0) {
        res.writeHead(400);
        res.end('RESPONSE NOT AN IMAGE');
      } else if (contype === 'image/jpeg') {
        handlerImageDownload(response, imagepath, function(size) {
          // Verifying the datasize
          if (size >= 10000) {
            // Save Mongoose schema to database with error handling
            imageSchema.save(function (err, saveImage) {
              if (err) {
                res.writeHead(500);
                res.end('ERROR WRITING TO DB');
              }
            });
          } else {
            res.writeHead(500);
            res.end('NOT ENOUGH DATA FOR IMAGE');
          }
        });
        // Everything went ok!
        res.writeHead(200);
        res.end('SAVED IMAGE, WROTE TO DATABASE');
      } else { // Everything else is responded with internal server error
        res.writeHead(500)
        res.end('INTERNAL SERVER ERROR');
      }
    });
  });
});


/* Get new comment */
router.get('/newComment', function(req, res, next) {
    var comment = { name: req.query.name, text: req.query.text, date: Date.now() };
      Image.findOneAndUpdate({'id' : req.query.id},
        {$push: {comments: comment}},
        {$safe: true, /*upsert: true*/},
          function(err, imageComment) {
            res.jsonp(imageComment);
          }
      );
});

module.exports = router;
```

A.1.4 config_template.js

```
var config = {};

config.mongodb = {};
config.image = {};

config.mongodb.ip = '';
config.mongodb.name = '';

config.mongodb.username = '';
config.mongodb.password = '';

config.image.frontpagelimit = 20;

config.image.folder = ''

config.image.topviews = 0;

module.exports = config;
```

A.1.5 views/layout.jade

```
octype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
    #logo
      a(href='/')
        img(src='/images/flackr_logo.png', alt="Flackr Logo")
    h2 The #1 website for sharing your images!
  body
    block content
```

A.1.6 views/index.jade

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
    #logo
      a(href='/')
        img(src='/images/flackr_logo.png', alt="Flackr Logo")
    h2 The #1 website for sharing your images!
  body
    block conten
```

A.1.7 views/image.jade

```
extends layout

block content

  h1 #{image.title}
  img(src='#{image.id}.jpg')
  h3 Published by #{image.publisher}
  p(style='white-space:pre;')
    | Views: #{image.views}
    | Date:  #{image.date.toDateString()}

  h3 Comments
  table
    each comment in image.comments
      tr
        td(style='white-space:pre;')
          | #{comment.name}
          | #{comment.text}
          | #{comment.date.toDateString()}
```

A.1.8 views/error.jade

```
extends layout

block content
  h1= message
```

A.2 Factory

A.2.1 Worker

Application code

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" Worker code for producing and consuming RabbitMQ messages
"""
import pika
import json
import time
import yaml

#open configuration file
with open("config.yml", 'r') as ymlfile:
    cfg = yaml.load(ymlfile)

#Declare rabbitmq connections
credentials = pika.PlainCredentials(cfg['rabbitUser'], cfg['rabbitPsw'])
parametersConsume = pika.ConnectionParameters(cfg['receiveFrom'], cfg['port'], \
                                    '/', credentials, retry_delay=15, \
                                    connection_attempts=5)
parametersProduce = pika.ConnectionParameters(cfg['produceTo'], cfg['port'], \
                                    '/', credentials, retry_delay=15, \
                                    connection_attempts=5)

#Start consuming connection
connectionConsume = pika.BlockingConnection(parametersConsume)
channelConsume = connectionConsume.channel()
#Declare durable queue
```

```
channelConsume.queue_declare(queue=cfg['queue'], durable=True)

#Function for getting current time
def timeStamp():
    """Function for generating timestamp"""
    stamp = time.ctime(int(time.time()))
    return stamp

#Function for producing messages
def produce(item):
    """ Function for producing a message to a message queue"""
    #Start producing connection
    connectionProduce = pika.BlockingConnection(parametersProduce)
    channelProduce = connectionProduce.channel()
    channelProduce.queue_declare(queue=cfg['queue'], durable=True)
    message = json.dumps(item)
    #Publish message to queue
    channelProduce.basic_publish(exchange='', routing_key=cfg['queue'], \
                                 body=message, \
                                 properties=pika.BasicProperties(\
                                    delivery_mode=2, \
                                    content_type='application/json'))
    connectionProduce.close()
    print " [*] " + timeStamp() + " Done producing"

#Function for consuming messages
def callback(ch, method, properties, body):
    """ Function for consuming messages from message queue"""
    print " [*] " + timeStamp() + " Accepted job"
    data = json.loads(body)
    sleep_tag = int(data["jobType"])
    time.sleep(sleep_tag)
    stamptime = timeStamp()
    data["stamp"].append({cfg['worker']:stamptime})
    channelConsume.basic_ack(delivery_tag=method.delivery_tag)
    print " [*] " + timeStamp() + " Done consuming, starting producing"
    produce(data)
    print " [*] " + timeStamp() + " Waiting for next job"

print " [*] " + timeStamp() + " Waiting for job"

channelConsume.basic_consume(callback, queue=cfg['queue'])
channelConsume.start_consuming()
```

YAML configuration file

```
---

receiveFrom: 192.168.200.129
```

```
produceTo: 192.168.200.141
port: 5672
queue: assembly
worker: worker1
rabbitUser: factory
rabbitPsw: factoryPass
...
```

A.2.2 Worker-end

Application code

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" Worker code for endpoint, writing to database"""
import pika
import sys
import json
import pymongo
from pymongo import MongoClient
import time
import yaml

#open config file
with open("config.yml", 'r') as ymlfile:
    cfg = yaml.load(ymlfile)

#Connect to database
try:
    dbCon = MongoClient('mongodb://' +cfg['databaseIP'] + ':' + \
                          cfg['databasePort'] + '/')
    db = dbCon[cfg['db']]
    posts = db[cfg['collection']]
    collection_exist = db.collection_names()
except pymongo.errors.ServerSelectionTimeoutError as err:
    print " [!]  Could not connect to DB: %s" % err
    print " [!]  No items will be written to DB!!"
    sys.exit()

#Establish pika connection to rabbitmq
credentials = pika.PlainCredentials(cfg['rabbitUser'], cfg['rabbitPsw'])
parameters = pika.ConnectionParameters(host='localhost')
connection = pika.BlockingConnection(parameters)

channel = connection.channel()

#declare a durable queue
channel.queue_declare(queue=cfg['queue'], durable=True)

def time_stamp():
    """ Function for getting timestamp"""
    stamp = time.ctime(int(time.time()))
    return stamp
```

```

#Function that consumes messages
def callback(ch, method, properties, body):
    """ Function for consuming messages from queue"""
    print " [*] " + time_stamp() + " Accepted job"
    data = json.loads(body)
    time.sleep(2)
    data["timeStop"] = time_stamp()
    if not collection_exist:
        data["id"] = 1
    else:
        try:
            get = posts.find({}, {"id":1, "_id":0})\
                    .sort([("_id", pymongo.DESCENDING)]).limi
                        t(1)
            for doc in get:
                try:
                    val = int(doc['id']) +1
                    data["id"] = val
                except KeyError:
                    print " [!] " + time_stamp() + \
                        "Error in database! ID field not found"
                    sys.exit()
        except pymongo.errors.ConnectionFailure as err:
            print " [!] " + time_stamp() + \
                    " Could not get ID from database!! %s" % err
            sys.exit()
    try:
        posts.insert_one(data)
        print(" [*] Wrote to database, ID: {}".format(data['i
            d']))
    except pymongo.errors.ConnectionFailure as err:
        print " [!] " + time_stamp() + \
                " Could not post to database! %s" % err
        sys.exit()
    channel.basic_ack(delivery_tag=method.delivery_tag)
    print " [*] " + time_stamp() + " Job finished"
    print " [*] " + time_stamp() + " Waiting for next job"

print " [*] " + time_stamp() + " Waiting for job"

channel.basic_consume(callback, queue=cfg['queue'])
channel.start_consuming()
```

YAML configuration file

```
---

databaseIP: 192.168.200.125
databasePort: '27017'
db: factory
collection: posts
queue: assembly
rabbitUser: factory
rabbitPsw: factoryPass
```

```

...
```

A.2.3 Finished Job Entry System

app.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" Simple web application
This application is built using Flask and PyMongo. It will query the database
and present the data on a simple website using a template.
"""

# Import
import yaml
from pymongo import MongoClient, errors
from flask import Flask, render_template


# Define
app = Flask(__name__)
maxSevSelDelay = 3000

# Config file setup
print " * Configuration file setup..."
with open("config.yml", 'r') as ymlfile:
    cfg = yaml.load(ymlfile)

# Database setup
print " * Database setup..."
try:
    dbCon = MongoClient('mongodb://' + cfg['databaseIP'] + ':' + \
            cfg['databasePort'] + '/', serverSelectionTimeoutMS=maxSevSelDelay)
    dbCon.server_info()
except pymongo.errors.ServerSelectionTimeoutError as err:
    print " * Could not connect to DB: %s" % err

db = dbCon[cfg['db']]
collection = db[cfg['collection']]


@app.route('/')
def index():
    ''' POST GET '''
    # Checks if special mode is enabled
    dogeMode = {'value' : cfg['dogeMode']}
    # Query database for entries
    query = collection.find().sort("id", -1).limit(cfg['frontpageLimit'])
    return render_template('index.html', entries=query, dogeMode=dogeMode)
```

```

@app.errorhandler(404)
def page_not_found(error):
    ''' Error handling for 404 '''
    return render_template('404.html', err=error), 404


if __name__ == "__main__":
    app.run(debug=True, port=5100, host='0.0.0.0')
```

requirements.txt

```
aniso8601==1.1.0
Flask==0.10.1
Flask-RESTful==0.3.5
itsdangerous==0.24
Jinja2==2.8
MarkupSafe==0.23
pymongo==3.2.2
python-dateutil==2.5.2
pytz==2016.3
PyYAML==3.11
six==1.10.0
Werkzeug==0.11.5
```

config.yml

```
---

databaseIP: 192.168.200.125
databasePort: '27017'
db: factory
collection: posts
frontpageLimit: 5000
dogeMode: False

...
```

template/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Finished Job Entry System - FJES</title>
        {% if dogeMode.value == true %}
                <link rel="stylesheet" href="{{ url_for('static', filename='index-doge.css') }}">
    {% else %}
        <link rel="stylesheet" href="{{ url_for('static', filename='index.css') }}">
        {% endif %}
</head>
<body>
        <center>
        <h1>Finished jobs</h1>
        </center>
        <hr>
        </hr>
        {% for entry in entries %}
        <p>
                <div id="page-wrapper">
                <h2> ID: {{ entry.id }} </h2>
                <ul>
                        <li> Name: {{ entry.name }} </li>
                        <li> Jobtype: {{ entry.jobType }} </li>
                        <li> Start: {{ entry.timeStart }} </li>
                        <li> Stop: {{ entry.timeStop }} </li>
                        <li> Stamps: </li>
                        <ul>
                                {% for stamp in entry.stamp %}
                                        {% for worker, time in s
                                            tamp.items() %}
                                                <li> {{ worker
                                                    }}: {{ time
                                                    }} </li>
                                        {% endfor %}
                                {% endfor %}
                        </ul>
                </ul>
                </div>
        </p>
        {% endfor %}
</body>
</html>
```

template/404.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Finished Job Entry System - 404</title>
        <link rel="stylesheet" href="{{ url_for('static', filena
            me='index.css') }}">
</head>
<body>
        <center>
        <h1>Page Not Found</h1>
        <center>
        <hr>
        </hr>
        <p>What you were looing for is just not here...</p>
        <img src="static/404-kitty.jpg">
</body>
</html>
```

A.2.4 entrypoint

app.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Simple web application.
```

```
This application built with Flask and RESTful works as an entryp
    oint to the
architecture. This will accept POST JSON, produce a job and put
    in on the
message queues with Pika.
"""
# Import
from flask import Flask, request, abort
from flask_restful import Resource, Api
import yaml
import time
import pika
import json

# Define
app = Flask(__name__)
api = Api(app)

# Config setup
with open("config.yml", 'r') as ymlfile:
    cfg = yaml.load(ymlfile)

# Route /job, POST only
class NewJob(Resource):
    """ API resource
    """
    def post(self):
        """     Take request data, puts into an
        object and passes it along to a queue.

        >>> curl -i -H "Content-Type: application/json" -X \
        POST -d '{"name" : "Ola Nordmann", "jobType" : 2}'
            127.0.0.1:5000/job
        HTTP/1.0 201 CREATED
        Content-Type: application/json
        Content-Length: 119
        Server: Werkzeug/0.11.5 Python/2.7.6
        Date: Tue, 12 Apr 2016 16:51:18 GMT

        {
            "jobType": 2,
            "name": "Ola Nordmann",
            "stamp": [],
            "timeStart": "Tue Apr 12 18:51:18 2016"
        }

        """

        # Not JSON, no name then 400 it.
        if not request.json or not 'name' in request.json:
            abort(400)
        # Create a variable current time.
        localtime = time.ctime(int(time.time()))
        # Get requested data and save it.
        json_dict = request.get_json(force=True)
        # Append stamp array and time variable.
        json_dict.update({"stamp":[], "timeStart":localtime})

        # Pika connection auth
        parameters = pika.ConnectionParameters(host='localhost')
        connection = pika.BlockingConnection(parameters)
        channel = connection.channel()

        # Declare the queue and use persistant messages.
        channel.queue_declare(queue='assembly', durable=True)

        # Pika message and delivery
        message = json.dumps(json_dict)
        channel.basic_publish(exchange='', routing_key='assembly', \
                body=message, properties=pika.BasicProperties(delivery_mode=2, \
                content_type='application/json'))
        print "Sent %r" % message

        #Close connection
        connection.close()
        return json_dict, 201

# Abstract RESTful resources
api.add_resource(NewJob, '/job')


if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0')
```

requirements.txt

```
aniso8601==1.1.0
Flask==0.10.1
Flask-RESTful==0.3.5
itsdangerous==0.24
Jinja2==2.8
MarkupSafe==0.23
pika==0.10.0
pymongo==3.2.2
python-dateutil==2.5.2
pytz==2016.3
PyYAML==3.11
six==1.10.0
Werkzeug==0.11.5
```

B Traffic generating script for Flackr

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
""" Script for generating traffic to flackr
"""
import urllib2
import urllib
import random
import os
import json
import getopt
import sys

"""Options:
 U = URL REQUIRED OPTION
 P = add new picture
 V = views + A all or R random(writes webpagetotal x2 random views)
 F = download "visit" frontPage
 C = comment + A all, or R random(gen number of comments(max 1/2 total)
                                    and then comments to random id)
 H = adds H_RANGE pictures, visits frontpage, comments to all
        visits all pictures once
 R = Either inserts picture, comments with opt R, views with opt R
      or visits frontpage F

 Example: ./trafficScript.py -U "http://192.168.200.136:80/" -P
 IMPORTANT: Remember '/' at the end!

 H_RANGE variable can be changed, depending on how many pictures you
 want to be added with option H """

FILEPATH = "/home/ubuntu/trafficscript/sentences.txt" #change for environment
TEMPDIR = "/run/shm/"
LINE_LIMIT = 2200
SENTENCE_LENGTH = 150
H_RANGE = 50  # Variable can be changed depending on wanted load.


FOUND_OPTION_U = False
OPTIONS, REM = getopt.getopt(sys.argv[1:], 'U:PV:FC:HR')
for alt, argum in OPTIONS:  #Check if arguments have -U
    if alt in '-U':
            URL = argum
            FOUND_OPTION_U = True

    if not FOUND_OPTION_U:
        print "NO␣URL␣GIVEN"
        sys.exit()

def get_webpage_items():
    """get number of pictures from webpage"""

    rand = get_int(0, 1000)  # Get random int for folder name
    folder = TEMPDIR + str(rand) #make folder path of tempdir and random int

    os.mkdir(folder)  # Make new folder
    os.chdir(folder)  # Change directory to folder
    os.system("wget␣-t␣2␣-T␣5␣-q␣"+URL) # Wget frontpage

    tempfile = folder+"/index.html"
    if os.path.isfile(tempfile):  # Check if file exists
        index_file = open(tempfile) # Opens it
        var = index_file.read() # Read entire file
        var2 = var.split()  # split file into lines
        search_item = "database:" # What to search for
        search_index = var2.index(search_item)
          # find the index of the searched item.
        database_count = int(var2[search_index+1])
          # Set next item after search item to be the count,
          # as the following item will be a number
        index_file.close() #close file
        os.system("rm␣-r␣"+folder)  # Remove folder and its contents
        return database_count
    else:
        print "Error,␣could␣not␣find␣or␣locate␣database_count"

def get_int(arg1, arg2):
    """ returns random int in range of arg1 and arg2"""
    return random.randint(arg1, arg2)

def is_sentence(txt):
    """ Function for check if sentence starts with not allowed characters"""
    return txt.startswith('<<') or txt.startswith('>>') or \
               txt.startswith(',') or txt.startswith('ELECTRONIC') or \
               txt.startswith('PROHIBITED')

def check_sentence_length(text):
    """ Function to check if sentence is within limits """
    if len(text) > SENTENCE_LENGTH:  # Check if sentence is longer than allowed
        temp = text[:SENTENCE_LENGTH].split()
```

```
          # Split sentence into separate words, but only up to allowed length
        new_sentence = "␣".join(temp[:len(temp)-1])
          # join the words from the list, -1 so last word isnt split
        return new_sentence    # return new sentence
    else:
        return text  # if not longer than allowed, return original


def front_page():
    """ Function, visit frontpage of flackr"""
    rand = get_int(0, 1000)  # Get random int for folder name

    folder = TEMPDIR + str(rand) # make folder path of tempdir and random int
    os.mkdir(folder)  # Make new folder
    os.chdir(folder)  # Change directory to folder

    os.system("wget␣-t␣2␣-T␣5␣-p␣-q␣"+URL)
      # Wget all elements from webpage inc css,logo,photos
    os.system("rm␣-r␣"+folder)  # Remove folder and its contents

    print "visited␣front␣page"

def get_random_name():
    """ Function for getting a random user name"""
                      # Downloads a json document with random user information
    response = urllib2.urlopen( \
      'http://api.randomuser.me/1.0/?nat=gb,us&inc=name&noinfo').read()
    parsed_json = json.loads(response)  # Parse the json into variable
    firstname = parsed_json["results"][0]["name"]["first"].title()
        # Select firstname and capitalize
    lastname = parsed_json["results"][0]["name"]["last"].title()
        # select Lastname and capitalize

    name = firstname + "␣" + lastname     # Add names together

    return name      # Return generated name

def get_random_title():
    """ Function for making a random title"""
            # Same procedure as previous function
    response = urllib2.urlopen( \
      'http://api.randomuser.me/1.0/?nat=gb,us&inc=name&noinfo').read()
    parsed_json = json.loads(response)
    temp = parsed_json["results"][0]["name"]["last"].title()
```

```
         # Select only lastname and capitalize
    title = "The" + "␣" + temp    # Make it sound like a title

    return title      # Return generated title

def new_picture():
    """ Function for adding new picture to website"""
    name = get_random_name()  # Get name and title
    title = get_random_title()
          # uses urllib module for opening up connection to webp
              age
          #  and add a new picture using the /insert route
    response = urllib.urlopen(URL+"insert/newUser?title=" + \
                              title + "&publisher=" + name \
                              + "&image=http://lorempixel.co
                                  m/1280/720/")
    #print "%s" % response
    print "added␣new␣picture"

def gen_views(option):
    """ Has 2 functions,visit all once,visit random, visit 1,
    visit a few many times. The idea is the same as explained
    in option A for all of them,  only difference is
    how many pages are visited. """

    items = get_webpage_items()
    opt_random = False
    if option == 'R':
        item_range = items * 2  # Visit all pages or random
        opt_random = True
    else:
        item_range = items
    for item in range(1, item_range + 1):
     # Range from 1 to nr of pictures,compensate +1 for range
        rand = get_int(0, 1000) # Generate random int for folder
            name
        if opt_random:
            item_id = get_int(1, items)
        else:
            item_id = item
        urlsite = URL+str(item_id) # Make URL
        folder = TEMPDIR+str(rand) #Make folder path
        os.mkdir(folder)      # Make folder
        os.chdir(folder)      # Change directory to folder
        os.system("wget␣-t␣2␣-T␣5␣-p␣-q␣"+urlsite)
                    # Download everything from webpage
        os.system("rm␣-r␣"+folder)  # Delete again
        print "View␣"+str(item_id)
    print "Visited␣"+str(item_range) + "␣number␣of␣sites"

def get_random_text():
    """ Get a random sentence from the sentences file"""
    find_line = False  # bool variable

    if os.path.isfile(FILEPATH):  # check if file can be found
        temp = open(FILEPATH)     # Open file
        text = temp.readlines()   # read all the lines
        while find_line is False:  # Until it finds a proper line
            line = get_int(0, LINE_LIMIT)  # Get random line
            if is_sentence(text[line]) is False:
              # Use is_sentence function to check if acceptable
                find_line is True    # Set bool value to true to stop while loop
                sentence = check_sentence_length(text[line])
                # Check if sentence is too long
                return sentence   # Return the sentence
        temp.close()
    else:
        print "Could␣not␣find␣file!!!"

def gen_comment(option):
    """ 2 functions, comment to all once, comment to random x times.
    All options use the same method,
    only difference being amount of comments """

    items = get_webpage_items()
    opt_random = False
    if option == 'R':
        item_range = items / 2
        opt_random = True
    else:
        item_range = items
    for item in range(1, item_range + 1):
     # comment to all or random, +1 is compensation for range.
        name = get_random_name()     # Get random name from function
        text = get_random_text()     # Get a sentence from the sentences file
        if opt_random:     # If option R, get random int for ID
            item_id = get_int(1, items)
        else:
            item_id = item  # Open up the url /insert route
        response = urllib.urlopen(URL+"insert/newComment?id=" \
                                   + str(item_id) +"&name=" + name \
                                   + "&text=" + text)
        #print "%s" % response
        print "generated␣comment␣to:␣" + str(item_id)

def random_operation():
    """ Function for doing one of the provided options
      randomly """
    random_number = get_int(0, 100)  # Gets int in provided range
    if random_number <= 25: # Depending on int it does one of the options
        new_picture()
    elif random_number > 25 and random_number <= 50:
        gen_comment(’R’)
    elif random_number > 50 and random_number <= 75:
        gen_views(’R’)
    elif random_number > 75 and random_number <= 100:
        front_page()

def main():
    """ Main menu, takes option as command line argument"""
    for opt, arg in OPTIONS:  # And based on input, calls required functions
        if opt in ’–P’:
            new_picture()
        elif opt in ’–F’:
            front_page()
        elif opt in ’–C’:
            if arg == ’A’ or arg == ’R’:
                gen_comment(arg)
            else:
                raise ValueError("wrong␣ARGUMENT!␣A␣or␣R!!!")
        elif opt in ’–V’:
            if arg == ’A’ or arg == ’R’:
                gen_views(arg)
            else:
                raise ValueError("Wrong␣argument!␣A␣or␣R!!!")
        elif opt in ’–H’:  # For now it adds H_RANGE users
            for _ in range(H_RANGE):
                new_picture()

            front_page()  # Visits frontpage
            gen_comment(’A’)  # comments to all
            gen_views(’A’)    # views all
        elif opt in ’–R’:
            random_operation()

if __name__ == ’__main__’:
    main()
```

C Traffic generating script for Factory

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" Script for generating traffic to rabbitmq service architecture
"""

from random import randint
import os
import getopt
import sys
import urllib2
import urllib
import json
import requests

"""Options:
 U = URL REQUIRED OPTION
 O = One new entry
 M = Many new entries

Example:
>>> ./scriptTraffic.py -U "http://192.168.200.129:5000/job" -M 2
{
     "jobType" : 5,
     "name": "Sam Neal",
     "stamp": [],
     "timeStart": "Mon Apr 25 16:14:27 2016"
}

{
     "jobType" : 2,
     "name": "Beverly Parker",
     "stamp": [],
     "timeStart": "Mon Apr 25 16:14:27 2016"
}

JobType can represent anything. For example 1 can be a car, 2 can be a plane
and so forth.

"""

# CONSTANTS
TEMPDIR = "/run/shm/"
TRACEBACK = 1

# Checking if URL (-U) is passed as argument
```

```
FOUND_OPTION_U = False
OPTIONS, REM = getopt.getopt(sys.argv[1:], 'U:OM:F')
for alt, argum in OPTIONS:
    if alt in '-U':
        URL = argum
        FOUND_OPTION_U = True

    if not FOUND_OPTION_U:
        print "NO␣URL␣GIVEN"
        sys.exit()


def get_random_name():
    """ Function for getting a random user name"""
    # Downloads a json document with random user information
    response = urllib2.urlopen('http://api.randomuser.me/1.0/?nat=gb,us&inc=name&noinfo').read()
    parsed_json = json.loads(response)  # Parse the json into variable
    firstname = parsed_json["results"][0]["name"]["first"].title()
    # Select firstname and capitalize
    lastname = parsed_json["results"][0]["name"]["last"].title()
    # select Lastname and capitalize

    name = firstname + "␣" + lastname     # Add names together

    return name     # Return generated name


def front_page():
    """ Function that visits the front page"""
    rand = randint(0, 1000)  # Get random int for folder name

    folder = TEMPDIR + str(rand) # make folder path of tempdir and random int
    os.mkdir(folder)  # Make new folder
    os.chdir(folder)  # Change directory to folder

    os.system("wget␣-t␣2␣-T␣5␣-p␣-q␣"+ URL)
    # Wget all elements from webpage inc css,logo,photos
    os.system("rm␣-r␣"+ folder)  # Remove folder and its contents

    print "visited␣front␣page"


def new_entry():
    """ Function to add a single new entry"""
    name = get_random_name()
    jobType = randint(1, 5)
    # Constructing the request
    payload = {'name' : name, 'jobType' : jobType}
    headers = {'Content-Type': 'application/json'}
    # Execute the request with error handling for general and 404
    try:
        req = requests.post(URL, data=json.dumps(payload), headers=headers)
    except requests.exceptions.RequestException as err:
        print_error_msg(err)
        sys.exit(TRACEBACK)
    else:
        if req.status_code == 404:
            print_error_msg(req.raise_for_status())
            sys.exit(TRACEBACK)
    # Print out the reply text
    print req.text
    return req.text

def many_new_entries(number):
    """ Function to add several new jobs, using previous functions"""
    # For number in range, add a new entry
    if number.isdigit():
        for i in range(int(number)):
            new_entry()
    else:
        print_error_msg("Passed␣variable␣is␣not␣a␣number.␣Exiting...")
        sys.exit(TRACEBACK)
    print "Done!␣Sent␣%s␣new␣entries." % number


def print_error_msg(err):
    """ Simple error print handler """
    print "Error␣message:␣%s␣" % err


def main():
    """ Main menu, takes option as command line argument"""
    # Depending on the input, choose an option
    for opt, arg in OPTIONS:
        if opt in '-F':
            front_page()
        elif opt in '-O':
            new_entry()
        elif opt in '-M':
            many_new_entries(arg)


if __name__ == '__main__':
    main()
```

D Disable Transparent Huge Pages

/etc/init.d/disable-transparent-hugepages

```
!/bin/sh
### BEGIN INIT INFO
# Provides:          disable-transparent-hugepages
# Required-Start:    $local_fs
# Required-Stop:
# X-Start-Before:    mongod mongodb-mms-automation-agent
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Disable Linux transparent huge pages
# Description:       Disable Linux transparent huge pages, to im
   prove
#                    database performance.
### END INIT INFO

case $1 in
  start)
    if [ -d /sys/kernel/mm/transparent_hugepage ]; then
      thp_path=/sys/kernel/mm/transparent_hugepage
    elif [ -d /sys/kernel/mm/redhat_transparent_hugepage ]; then
      thp_path=/sys/kernel/mm/redhat_transparent_hugepage
    else
      return 0
    fi

    echo 'never' > ${thp_path}/enabled
    echo 'never' > ${thp_path}/defrag

    unset thp_path
    ;;
esac
```

E MongoDB configuration file

```
# mongod.conf

# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/

# Where and how to store data.
storage:
  dbPath: /var/lib/mongodb
  journal:
    enabled: true
#  engine:
#  mmapv1:
#  wiredTiger:

# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

# network interfaces
net:
  port: 27017
  bindIp: 0.0.0.0

#processManagement:

security:
  authorization: enabled
#operationProfiling:

#replication:
  replSetName: rs0
#sharding:

## Enterprise-Only Options:

#auditLog:

#snmp:
```

F Nginx configuration file

File /etc/nginx/sites-available/default

```
proxy_cache_path /data/nginx/cache/ levels=1:2 keys_zone=backcache:8m max_size=50m;
proxy_temp_path /data/nginx/cache/tmp;
proxy_cache_key "$scheme$request_method$host$request_uri$is_args$args";
proxy_cache_valid 200 302 5s;
proxy_cache_valid 404 1m;

upstream flackrapp {
  server 192.168.200.104:3001;
  server 192.168.200.112:3001;
}

server {
  listen 80;
  listen [::]:80;


  location = / {
    proxy_cache backcache;
    proxy_cache_bypass $http_cache_control;
    add_header X-Proxy-Cache $upstream_cache_status;

    proxy_pass http://flackrapp;
   }
  location / {
    proxy_pass http://flackrapp;
  }

}
```

G User manuals

G.1 Flackr

The user manual is supposed to give a brief and simple setup guide to get Flackr up and running. It is basically a shortened and compressed version of the Implementation section. The user manual is designed for the students. The guide will go through setting up the node.js webservers, MongoDB database, GlusterFS network filesystem and Nginx. If you want to read more about the technologies used to deploy Flackr, go to section 3.1 Technologies.

To deploy a complete Flackr system you will need the following servers:

- 1-3 Node Webservers
- 1 MongoDB
- 2 x GlusterFS
- 1 Nginx

G.1.1 Webserver

```
1. sudo apt-get install npm git nodejs-legacy
2. Clone the repository
3. Change to the directory
4. Checkout tags/<version number>
5. sudo npm install -g
6. cp template_config.js config.js
7. Fill out the variables
8. sudo npm install pm2 -g
9. sudo npm install pm2-logrotate
10. Start the application with logging and autostart with the following commands
  10.1. pm2 start app.js --name "flackr"
  10.2. sudo pm2 startup ubuntu
  10.3. sudo pm2 logrotate -u ubuntu
  10.4. Logging can be verify by checking /etc/logrotate.d/pm2
```

G.1.2 MongoDB

```
1. sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
2. echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
3. sudo apt-get update
4. sudo apt-get install -y mongodb-org
5. Copy disable-transparent-hugepages file from appendix
6. sudo chmod 755 /etc/init.d/disable-transparent-hugepages
7. sudo update-rc.d disable-transparent-hugepages defaults
8. Add the following to /etc/mongod.conf
```

```
         # network interfaces
            net:
            port: 27017
            bindIp: 0.0.0.0
9.  mongo
    10. use admin
    11. db.createUser(
        { user: "userAdmin",
           pwd: "userAdminPassword",
           roles: [{role: "dbOwner", db: "admin" }]
        }
        )

    12. db.createUser(
        { user: "flackr",
           pwd: "flackrPassword",
           roles: [{role: "readWrite", db: "flackr" }]
        }
        )
    13. Exit
14. Add the following to /etc/mongod.conf
    security:
        authorization: enabled
15. sudo service mongod restart
16. mongo -u "userAdmin" -p "userAdminPassword" --authentication
   Database "admin"
    17. >use admin
        > db.getUsers()
        exit

Replication
1. Add the following to /etc/mongod.conf
    replication:
        replSetName: rs0
2. rs.initiate() # Start replication set
3. rs.conf()     # Configure replica set
4. rs.add('ip add slave')
Gives:
    Return: {"ok":1}
5. rs.status()
```

G.1.3 GlusterFS

All servers involved with GlusterFS:

```
1. sudo apt-get update
2. sudo apt-get install software-properties-common
3. sudo apt-get install attr
 Then add the community GlusterFS PPA:
4. sudo add-apt-repository ppa:gluster/glusterfs-3.5
```

GlusterFS servers:

```
1. sudo apt-get update
2. sudo apt-get install glusterfs-server
3. sudo apt-get install xfsprogs
```

```
attach storage volume in skyhigh, and reboot servers.
Do the following on GlusterFS server01:
3. sudo gluster peer probe ip-address-of-gluster02
Both hosts(NOTE: the folder name is different on each server)
4. ls /dev/disk/by-id/
    example id: virtio-b6dde619-feed-455b-a
5. mkfs.xfs -i size=512 /dev/disk/by-id/virtio-b6dde619-feed-455
    b-a
Remember to do this on both servers(glusterbrick01 and glusterbr
    ick02):
6. sudo mkdir -p /data/glusterbrick01
7. sudo mount /dev/disk/by-id/virtio-b6dde619-feed-455b-a /data/
    glusterbrick01
On master host:
8. sudo gluster volume create imagestorage replica 2 transport t
    cp ip-address-of-gluster01:/data/glusterbrick01 ip-add-of-gl
    uster02:/data/glusterbrick02 force
9. sudo gluster volume start imagestorage
```

```
Clients:
1. sudo apt-get update
2. sudo apt-get install glusterfs-client
3. sudo mkdir -p /data/imagestorage
4. sudo mount -t glusterfs ip-add-of-gluster01:imagestorage /dat
    a/imagestorage/
5. sudo vi /etc/fstab
    ip-add-of-gluster01:/imagestorage /data/imagestorage gluster
        fs defaults,nobootwait,_netdev,backupvolfile-server=ip-a
        dd-of-gluster02,direct-io-mode=disable 0 0
6. sudo chown ubuntu:ubuntu /data/imagestorage
```

G.1.4 Nginx

```
1. sudo apt-get update
   sudo apt-get install nginx

2. sudo mkdir -p /data/nginx/cache
   sudo chown <username> /data/nginx/cache
   sudo chmod 700 /data/nginx/cache

3. Add content from configuration file in the appendix.
```

G.2 Factory

The following user manuals requires a minimum of 6 servers.

- 1 Entrypoint server
- 2 Workers
- 1 Queue server
- 1 Safe server
- 1 Database server

G.2.1 Application servers

Has to be installed on all servers, except the database server. Note that on the Queue server, ONLY RabbitMQ has to be installed.

G.2.2 RabbitMQ

```
echo 'deb http://www.rabbitmq.com/debian/ testing main' |
    sudo tee /etc/apt/sources.list.d/rabbitmq.list

wget -O- https://www.rabbitmq.com/rabbitmq-signing-key-public.as
    c | sudo apt-key add -

sudo apt-get update
sudo apt-get install rabbitmq-server

sudo rabbitmqctl add_user factory factoryPass
sudo rabbitmqctl set_user_tags factory administrator
sudo rabbitmqctl set_permissions factory ".*" ".*" ".*"

sudo service rabbitmq-server restart
```

G.2.3 Python packages, Nodejs and PM2

```
sudo apt-get install -y python python-pip git

sudo apt-get update
sudo apt-get install -y npm nodejs-legacy

sudo npm install pm2@latest -g
```

G.2.4 Safe server

Install the database flask application.

```
Pull/clone FJES from git
cd the cloned folder
sudo pip install -r requirements.txt
chmod +x app.py
Edit the configuration file:
  databaseIP: IP address to server configured with MongoDB
  databasePort: Port that the database uses, default: 27017
  db: Database name
  collection: Collection name
  frontpageLimit: Limit the number of entries on front page, def
     ault: 5000
  dogeMode: {True | False}

pm2 start app.py --name FJES
```

Install the worker application.

```
Pull/clone workerEnd app from git
cd folder
sudo pip install -r requirements.txt
chmod +x workerEnd.py
Edit configuration file:
    databaseIP: IP address of MongoDB server
    databasePort: Port of database server, default: 27017
    db: Database name
    collection: Collection name
    queue: Queue name
    rabbitUser: RabbitMQ user
    rabbitPsw: RabbitMQ password
pm2 start workerEnd.py --name workerEnd
```

G.2.5 Entrypoint app server

```
Pull/clone from git
cd the cloned folder
sudo pip install -r requirements.txt
chmod +x app.py
pm2 start app.py --name entrypoint-app
```

G.2.6 Worker application servers

Note that the configuration file is individual for each worker. Make sure IP addresses are correct. Draw diagram if uncertain.

```
Pull/clone worker application from git
cd cloned folder
chmod +x worker.py
edit configuration file:
    receiveFrom: server to fetch messages from
    produceTo: server to send messages to
    port: RabbitMQ port, default: 5672
    queue: Rabbit queue to be used. (same on all)
    worker: worker0x ( X is the number of the worker, starting a
        t 1)
    rabbitUser: RabbitMQ username
    rabbitPsw: RabbitMQ password

pm2 start worker.py --name workerX (x is worker number)
```

G.2.7 MongoDB

```
1. sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --
   recv EA312927
2. echo "deb␣http://repo.mongodb.org/apt/ubuntu␣trusty/mongodb-o
   rg/3.2␣multiverse" | sudo tee /etc/apt/sources.list.d/mongod
   b-org-3.2.list
3. sudo apt-get update
4. sudo apt-get install -y mongodb-org
5. Copy disable-transparent-hugepages file from appendix
6. sudo chmod 755 /etc/init.d/disable-transparent-hugepages
7. sudo update-rc.d disable-transparent-hugepages defaults
8. Add the following to /etc/mongod.conf
        # network interfaces
           net:
           port: 27017
           bindIp: 0.0.0.0
9. sudo service mongod restart
```

G.2.8 Monitoring

```
Munin Node:

```

```
sudo apt-get update
sudo apt-get install -y munin-node
sudo nano /etc/munin/munin-node.conf
Change this line so it matches with the munin master node IP: al
    low ^127.0.0.1\$
cd /usr/share/munin/plugins
sudo git clone https://github.com/ask/rabbitmq-munin.git
sudo cp rabbitmq-munin/* .
sudo munin-node-configure --shell

sudo ln -s '/usr/share/munin/plugins/rabbitmq_connections' '/et
    c/munin/plugins/rabbitmq_connections'
sudo ln -s '/usr/share/munin/plugins/rabbitmq_consumers' '/etc/m
    unin/plugins/rabbitmq_consumers'
sudo ln -s '/usr/share/munin/plugins/rabbitmq_messages' '/etc/mu
    nin/plugins/rabbitmq_messages'
sudo ln -s '/usr/share/munin/plugins/rabbitmq_messages_unacknowl
    edged' '/etc/munin/plugins/rabbitmq_messages_unacknowledged'
sudo ln -s '/usr/share/munin/plugins/rabbitmq_messages_uncommitt
    ed' '/etc/munin/plugins/rabbitmq_messages_uncommitted'
sudo ln -s '/usr/share/munin/plugins/rabbitmq_queue_memory' '/et
    c/munin/plugins/rabbitmq_queue_memory'

sudo munin-node-configure --suggest (sjekk om rabbit_* er yes|ye
    s)
sudo service munin-node restart
sudo nano /etc/munin/plugin-conf.d/munin-node
Add the following:

[rabbitmq_connections]
user root

[rabbitmq_consumers]
user root

[rabbitmq_messages]
user root

[rabbitmq_messages_unacknowledged]
user root

[rabbitmq_messages_uncommitted]
user root

[rabbitmq_queue_memory]
user root

sudo service munin-node restart

You will notice an critical error on the rabbitmq_queue_memory,
    this is nothing important and you can fix it by editing the
    plugin on all nodes:
sudo nano /etc/munin/plugins/rabbitmq_queue_memory
```

```

Comment out the follwing 2 lines:
         #QUEUE_WARN=${queue_warn:-10000}
         #QUEUE_CRIT=${queue_crit:-20000}


sudo service munin-node restart
```

H Verification from Kyrre

Fra: Kyrre Begnum
Til: Stian Svalstad
Cc: Arnt-Helge Nilsen Øyan , Sigve Næss , Stian Asphoug Svalstad
Tittel: Re: Møte rundt arkitektur nr 2
Dato: 08-04-2016 12:39
Jeg bekrefter herved at forskjellige versjoner ikke er blir en del av spesifikasjonen til arkitektur 2.

I A JSON example from https://randomuser.me

```json
{
    "results": [
        {
            "user": {
                "gender": "female",
                "name": {
                    "title": "miss",
                    "first": "sophie",
                    "last": "snyder"
                },
                "location": {
                    "street": "5329␣high␣street",
                    "city": "ripon",
                    "state": "cumbria",
                    "zip": "T7␣5AG"
                },
                "email": "sophie.snyder@example.com",
                "username": "whitedog998",
                "password": "cashmone",
                "salt": "nMXhFzRZ",
                "md5": "1474f06ead8dd8ef423ffad911a2dc7d",
                "sha1": "2c608bbcd1ff88ed80335db7510e6583a676fa24",
                "sha256": "a4b323b3f922e24e1a8d5a85cc621f6d29c34efd289c0f3aaab9dd1d782013aa",
                "registered": 1114644312,
                "dob": 271565566,
                "phone": "016973␣61866",
                "cell": "0784-561-342",
                "NINO": "PK␣35␣13␣36␣Y",
                "picture": {
                    "large": "https://randomuser.me/api/portraits/women/34.jpg",
                    "medium": "https://randomuser.me/api/portraits/med/women/34.jpg",
                    "thumbnail": "https://randomuser.me/api/portraits/thumb/women/34.jpg"
                }
            }
        }
    ],
    "nationality": "GB",
    "seed": "efddc3c6b8e3bf8d08",
    "version": "0.8"
}
```

J Meeting logs

This is a compilation of all meetings throughout the bachelor thesis in Norwegian.

2016-01-11 Kjapt møte med ErikH

Avtale møtetider - Mandag 13:30

Tolkning av oppgaven (for å få et annet perspektiv) - Amazon reference architecture

Miljø vi kan jobbe på (SkyHigh) - SkyHigh burde gå greit - Brukerkontoer fra neste uke - Hvor mye kapasitet trenger vi?

Alternativer til LAMP stack (MEAN: MongoDB, Express.js, Angular.js, Node.js) - Ønsket at Erik kunne sette seg inn litt mer.

Oppetid-system - Ønskelig at Erik også setter seg inn litt her. - Se hvordan Kyrres løsning brukes per dags dato mot

Ta kontakt med Kyrre asap.

Spørsmål rettet til Kyrre: Forstå hvordan genering av trafikk fungerer per dags dato (gjenbruk i den grad vi kan?)

2016-01-15 Første møte med Kyrre

Oppmøtte: Arnt-Helge, Sigve, Stian, Kyrre. K108

OBS: Dette er en veldig tl;dr av noe av det Kyrre sa.

Nettside som f.eks Galleriside eller noe med rikt tekstdokument (qQuotes (replikker), hente fram forskjellige quotes.)

Tenk kule nettsider, løsninger. Tenk på hva vi bruker per dags dato (reddit, pinterest, flickr, etsy mm).

Bruke API'en vår til å legge inn data. Samme måte som bookface gjør nå.

CouchDB

Standardiserer. Glusterfs, memcached osv.

Kult fra Kyrre: RabbitMQ, ZeroMQ, osv. Putter på kø, jobbgreie som kverner på køen, bra skalering,

Muligens test på slutten mot klassen IMT3441.

Vi burde brainstorme, få ned noen idéer på papiret. Dermed gruble litt på disse. Send forslagene til Kyrre og snakk med han. Ha en åpen dialog med Kyrre hele veien. Positiv til møte hver 3-4 uke.

2016-01-18 Erikmøte

Tilstede: Arnt, Stian og Erik.

* Invite Erik med leserettigheter til ShareLatex prosjektet. Worst case scenario sende pdf. (evt andre enn webtjenester) * Sigve: Spesielt ansvar om at rapporten at vi har tenkt sikkerhet. Sikkerhetsaspektet skal ikke være et hinder. * Skriv ned kravspesifikasjon for SkyHigh bruker, send til Eigil og Erik.

2016-01-25 Erik

Tilstede: Arnt, Erik, Stian

Generell status Prosjektplan, verktøy, skal vi ha "fasit" utrulling med Puppet på en måte. Referens Erik Hjelmås (bare skriv ned)

* Generelt om progresjonen i prosjektplanen (verktøy) Selv om den er nesten ferdig, så gjør vi det lurt i å fortsette med andre ting. * Spørsmål om vi skal ha en slags "fasit". Automatisk utrulling i Puppet eller lignende.

(Off-topic) (Greit å skrive Erik som referens på CV, trenger ikke å spørre.) (Ikke ta med alt tull på CV'en, kjør heller på med generell utdanning og erfaring, deretter fyll på med hva man er stolt av/gjort/lignende). (Karaktersnitt, studiepoeng)

2016-02-01 Møte

TIlstede: Arnt, Sigve, Stian

Møte med Kyrre

Hva har dere gjort i forrige uke? Sigve: Prosjektrapport, les igjennom tidligere rapporter anbefalt av Tom, copy-paste forprosjekt, mongoose + bilder Stian: Mongoose, mongodb, bilder, mongoose + gridfs, flackrbilder mappen Arnt: Bloggen, express, bilder, mongoose

Skjerpings på klokken. Mulig endre avtalt møtetid for å jobbe sammen? Stian: Høre med kjærringa om når det går ann å møtes. Forventer tilbakemelding i morgen. Sigve: Møtes 10-18 Arnt: Samme for meg

Database og bilder (TENK UTENFOR BOKSEN!) - Database skal lagre data - Filsystemer er laget for å håndtere filer. Bilder er filer. - Klokt valg å lagre bildene i NoSQL? - GridFS er laget for å lagre STORE bilder For hvert bilde trenger vi en fs.file og en fs.chunk. Vi trenger chunks. Men om file er veldig små, får vi overheaden uten å egentlig dra fordel med chunksene. Bad bad bad performancewise.

Versjonsbasering - Ha dette i tankene.

User stories, epics, flows. - Views på bilder - Trending - Comments

Møte med Erik i dag - Vise hvor langt vi er kommet - Hvilken adresse vi skal invitere han til de forskjellige verktøyene - Yolo

Eventuelt

2016-02-02 Erik

Utsatt av Erik Tilstede: Arnt, Erik og Sigve

Kommentar forprosjekt: Framdriftsplanen er grei så lenge dere hoder til hva man skal gjøre. Risiko da den er lite spesifisert. Fornuftig med lik tid på begge deler? (Nei

Alternative idéer fra Erik: RT, CRM, redmine, (nei, vi vil ikke bruke disse.

Avtale møte med Kyrre, avklaring på hva som skal lages. Nøkkelord: lage nettside selv blant annet

Oppskriften av rapporten, hvor mye fokus på rapporten. Ikke nok med at den skal brukes som noe Kyrre kan lage et utdanningsopplegg på, men vi skal få karakter basert på dette. Viktigste: produsere noe som Kyrre kan bruke.

Ressurspersoner: Rune Hjeldsvold, Mariuz, Øyvind Kolloen, Gerardo, Simon

Tenk som Kyrre!!!

2016-02-10 Gruppemøte

Kjapt statusmøte. Tilstede: Arnt, Sigve, Stian

Info: Trello - Må brukes mer aktivt. Lite oppdateringer. Er vi stuck?

Info: Blogg - Arnt har tatt ansvaret. Oppdaterer bloggen en gang i uken basert på trello.

Sigve: ShareLatex - lag nytt prosjekt

Arnt: Møte med Kyrre - Sender e-post... igjen. :D Fortelle hvilken idéer vi har gått for. - Vise fram hva vi har gjort hittil. - Hva den er bygd opp av - Hva vi tenker på. - Hvordan han generer trafikk til Bookface (navn + bilde, kommentar, poster)!!

Info: Nettsiden

Status: MongoDB + bilder - Hente ut bildet

2016-02-12 Kyrremøte rundt brukerscript

Tilstede: Sigve, Arnt, Kyrre

webuse.perl 3 (4) options 1. Ny bruker 2. Ny post 3. Ny kommentar (4. last ned forside)

1. Simulere trafikkflyten API for å hente navn URL avatardb
2. Forsiden Plukker 1 Poster i vedkommenes navn
3. Laste ned forsiden Plukker et navn Plukker et annet navn Poster noe der
4. Laster ned forsiden

Kyrre skulle grave fram perl scriptet sitt (Må sikkert minne han på om dette... :D)

———-

Skriptet skal genere bildet lokalt, for å sende det over. Da vil du få trafikken for å sende ting over. Tenk at en bruker har et bilde på mobilen han vil laste opp sammen med sin brukerinformasjon og dato.

Problemer å ha javascript å browseren er å simulere det etterpå (Kommentar Arnt: ikke at vi har så veldig mye javascript på browsersiden uansett..., her er det mer at alt rundt (Node og Express primært) er skrevet i JS.

Ikke tenk en interaktiv nettside. Hvis det ikke kan puttes i et skript så er det ikke verdt å gjøre. Derfor ikke tenk på å klikke videre inne på /pictures sånn som vi vurderte

Lag noen flows, epics, (user stories) . og når dette tenkes så tenk at det skal automatiseres: * Hjemmesiden, hente bildet som man kan embedde inn på deres hjemmeside. * Antall views på bildene /:picture * Trender (reddit hug of death, slashdotta)

Dette er noe jeg og Sigve diskuterte i etterkant

Versjonsbasering v1: bilde i mongodb v2: migrering av bilde fra mongodb til en image server som gir url. url skal være i mongodb istedenfor bilde data.

2016-02-15 Gruppemøte

Tilstede: Arnt, Sigve, Stian

Hva har dere gjort i forrige uke? Stian: Counters (telle opp ID'en) som ikke fungerte, Mongoose auto increment som ikke fungerte, samarbeid med Arnt om å lagre bildene/hente fram. Sigve: Bilder lagret lokalt, begynte å skrive i dokumentasjon Arnt: Samarbeid med Stian om bilder, views fungerer, legge inn data i databasen ved bruk av URL, fikse en måte å telle opp ID'en

Versjonsbasering: Limit antall bilder på forsiden.

Møte med Erik i dag: Ingenting spesielt.

Hva skal gjøres denne uken: Stian: Legge inn comments Arnt: Forsiden, ahref. Ta ting sånn som det kommer. Sigve: Rapporten fram til tirs/onsdag. Deretter se på script.

Eventuelt: Onsdag blir halv dag. Stian og Arnt har eksamen torsdagen og vil veldig gjerne jobbe litt med det. 08:00-12:00.

2016-02-15 ErikH

Tilstede: Arnt, Stian, Sigve, Erik

Spørre Rune Hjelmås og Øivind Kolloen angående lagring av bilder

Verdien i oppgaven ligger i knytningen til skripting, trafikkgenerering, alt rundt selve applikasjonen.

Hvordan skal Kyrres bruke dette i kurset.

Google lignende arkitekturer, normal ville bruke noe ala dette, men vi trenger å lage et system er enklere, transparent, laget for å virke sammen med eksisterende teknologier som Kyrre bruker allerede nå.

erik.hjelmas gmail for google ting

2016-02-22 Gruppemøte

Tilstede: Arnt, Stian, Sigve

Versjonsbasering (glusterfs implementasjon: istedenfor å ta backup av mappen bilder så kan du få redundans ved bruk av glusterfs)

2016-02-29 Gruppemøte

Tilstede: Stian, Sigve

Kort tid igjen.

Hva gjorde dere forrige uke og hva er status? Arnt: Limit på antall bilder, grid på forsiden, error routing i hytt og pine, comment section css, layout for :id, endret rutingen fra /image/:id til /:id, litt json og python med Stian Stian: Mandag: Routingen Insert comment. Ryddet opp skjemaet. Ons: Lest og forstått kyrre script. Started opp git repo for script. Fredag,Lørdag,Søndag: Scripting. Timer jobbet ca 25. Status: Script nesten ferdig. Må testes. Sigve: man/tirs: introduction/system req. ons: lese opp på script tors: document structure

Versjonsbasering For nå er det det som står i Trello. Liten brainstorm?

Lage en enkel installasjonsguide som kan utdypes på et senere tidspunkt - Lagt til noen punkter på trello under generelt. Greit å ha som huskepinne, skulle kanskje hatt dette tidligere..... :D

Skrive, skrive, skrive.

2016-03-04 Kyrre

Tilstede: Arnt, Sigve, Stian, Kyrre

* Tanker rundt versjoner Versjonene tilknyttet arkitektur, går greit Problemer med at man kan gå rett til siste Bra med LB og content distrbution

Feature flags - Skru på memcache - Presentasjon 2009 devops (Sigve har den kanskje), Flickr, dark launch (introdusere et vindu i flickr, hvor du kan se aktiviteten til dine venner), feature som simulerer trafikken men kaster det bort. Noe som gjør ekstra spørringer, klienten gjør det men viser det ikke, gjør det og viser det. - Introdusere et eller annet driftsproblematikk: udefinerte variabler og lignende, spyr ut feilmeldinger,

* Om neste arkitekturen Noe å jobbe videre med Fabrikk Dokument (human readable, json) som går igjennom flere "maskiner" Samlebånd Generisk rammeverk: en kø,

flere fabrikkstasjoner, tar et dokument, prosesserer (aktiv: bruker cpu eller io/passiv: timeout), stamp, sender videre til en annenn kø, api

Konfigurasjon: en stasjon med veldig mange arbeidere med forskjellige køtid. Trello CLI hvor du kan dytte kortene framover. En visuelt representasjon. Et punkt må en person dytte den over i neste køen.

Politisk faktor Salg av mobilabonnement: salg (oppretter tilbudet) -> oppretter brukerkonto -> pakke sammen pakken

* Overlevering Tagger, og ikke brancher. Lage kronologisk rekkefølge på push. Hvordan dette blir brukt, oppgaver og lignende, ta utgangspunktet at en person kan ta i bruk dette, hvordan sjekke at dette fungerer (liste o.l.).

* Egenutviklet Eksisterende er så komplisert, endring fra semseter til semester, keep it simple stupid, se matrix (se hvordan alt er bygget opp)

2016-02-29 The gruppe guys

Tilstede: Arnt, Sigve, Stian

(Brukte mye tid på å få inn bilder i databasen)

Hva gjorde dere forrige uke og hva er status? Arnt: Node app i produksjon, nginx, glusterfs, testing. Stian: Script + debugging, writing python standard . PM2. Testing. Remove default URL from python script is the only thing left. Sigve: Rapporten, dokumentstruktur, mongodb

Sammendrag fra møtet med Kyrre - What to do next - Limited time Dark launch: v2 query, dumper data. v3 implemetasjon, enkel entry på forsiden "Bilde nr x har mest views". Logging: Work in progress. Fant en node modul som kan sjekkes. Driftproblematikk: Udefinerte variabler, spyr ut feilmeldinger og lignende.

Møte med Erik: Dark lanuch Logging Finpuss

2016-03-07 ErikH

Møte med Erik: versjonsbasering, skript

Bør ha møte med kyrre på fredag. Planlegg godt, mulig vi ikke ser kyrre før etter påske. Greit å vite at vi får nok info fra kyrre slik at vi kan jobbe godt fremover den neste mnd. Spør om versjonene. Legg planer for neste case.

Pass på å skrive kvalitetskode!! Python og node.js Skrive i rapporten, at vi har fulgt kodestandarer. Sikkerhets testing verktøy? Demonstere for sensor at vi har holdningen at koding skjer med kvalitet. Ikke bare for å få ting til å virke :D

Sjekke for vanlige programmeringsfeil i python.

2016-03-07 Veiledningsmøte

Tilstede: Arnt, Sigve, Stian, Erik

Sende utkast til Erik etter onsdag 16/03

2016-03-14 Møte med Erik

Tilstede: Arnt, Sigve, Erik

Skrive, skrive, skrive.

Trafikkgenerering er viktig, neste så det skulle ha hatt et eget kapittel på samme måte som teknologi, design og lignende.

2016-03-30 Gruppe

Tilstede: Arnt, Sigve

Arnt: Blogg, ZeroMQ vs RabbitMQ, sett på hvordan fabrikker er bygd. Sigve: Lesing av prosjektoppgaven.

Fra mandagen 11/04 SKAL Toggl brukes framover i to uker (med forebehold om at Arnt og Stian har eksamen i Ruting og Svitsjing på mandagen 07/04)

2016-04-04 Gruppemøte

Tilstede: Arnt, Sigve, Stian

RabbitMQ, planlegging, teknologier, design,

E-post til Kyrre - Blir gjort etter møte med Erik

Møte med Erik - Innholdet i e-posten til Kyrre - Sende PDFen til Erik etter møtet, trenger bare en leserunde først - Driftsproblematikk - Sikkerhet?

2016-04-04 Erik

Tilstede: Erik, Arnt, Sigve, Stian

Status * Skriving - ok, mangler punkt testing og user guides. En dag med dette så burde det være good. Erik sjekker ShareLatex. * Arkitektur - går bra

Kyrre - Skala, driftsproblematikk, få noe mer ut av Message Queue, versjonsbasert, mulig konfigurasjonsverktøy, Trello CLI. - Erik hadde ikke så mye å tilføye...

Driftsproblematikk som vi kan introdusere med denne arkitekturen - Ingenting konkret akkurat nå

Sikkerhet - Blir "vi" (Sigve) spurt om spesifikt sikkerhetsaspekt - Generell sikkerhet/Bevissthet rundt dette - Hvordan er prosjektet relevant for informasjonssikkerhet - En release til en annen forbedrer sikkerheten - Sertifikater for å kommunisere mellom køer - Redundans - Aksesskontrol - DoS (lett å fylle opp køen, ressurser) - Greit å aksepterer risikoen så lenge man vet om det, men hvis man ikke vet om det - Vil studentene som bruker denne arkitekturen få et dårlig forhold til sikkerhet? Reflekter!

Relatere dette til en case, som også er IT orientert - E-post - Digitalt postkontor - PDF Asynkron i forhold til flere applikasjoner Ie OpenStack booter flere instanser med parametere Tenk over use case

2016-04-08 KyrreMøte!

Viktig med litt variasjon, flackr, bookface som webapps, og et annet

Fabrikk

Skalering: - Flaskehals - Oversikt/overvåkning, relevant men ikke bruk icinga/nagios. Ha heller en enkel nettside med rabbitmq list queues og f.eks 100 siste jobber. - 1x samlebånd, Lee Shore stopper en server så vil hele samlebåndet stoppe, tenk redundans

Driftsproblematikk: - Generell overvåking ok, mindre jo bedre. - Konfigurasjonsverktøy, ikke bruk - Flaskehals - Single point of failure - Kompleksitet, like navn, klare skiller, overblikk, tegne hvordan det ser ut -

Versjonsbasering: - Få det skriftlig at vi kan avvike fra oppgaven på arkitektur nr 2. Se e-post.

Teknologimessing: - bare til info

Undervisningsverktøy/Use case - det skal brukes i undervisning - Use case * Kjøper abonnement, programmerer GSM, kviterer ut, salgsavdeling opprette business og fakturering, pakke ned og sende det. Saksflyt, saksbehandling. * Dokumentprossesering på et sykehus, kjøre MR resultater, lagt i en kø, generere high resolution, scanne (image recognation) * Snapchat, bilder. Får inn bildet, gjøre endringer, lagre dette, distribure dette. - Der vi bruker en kø.

To køer - En jobbkø - En svarkø

JSON objektet har start tid, for å se hvor lang tid det tar

2016-04-11 Gruppemøte

Tilstede: Arnt, Sigve, Stian

* Toggl

* Arnt reiser sannsynligvis i løpet av 2 uker

* Messenger, ubytte av melon? 2 mot 1. - Byttet ut til et kvinnfolk

* Sikkerhet - Stort sett done, for nå

Framover Mandag 11/04: Deploye Flackr. Arnt: Flask, restful API med POST Stian: Applikasjonen skal kjøre på worker. Feilsøking. Sigve: Monitoring

Monitoring: - Enkel nettside - Relevant data

Møte med Erik - Use case - Nedskalering - Ingen versjonsbasering

2016-04-11 Erik

Tilstede: Arnt, Stian, Sigve, Erik

Hvordan går Fabrikk?

Testing av Flackr - teste med klassen, får se hva som skjer

Use case - Anbefales å ta det med i rapporten eller når vi presenterer det så binde det opp mot et use case for å bedre kunne presentere.

Hvis noen kan frigjøres til skrive, så gjør gjerne det...

Sigve, sikkerhet - skriv gjerne på begge caser - hva, hvordan vil denne kunne angripes hvis den var sluppet ut i den virkelige verden - anta at noen vil alltid angripe det

Til neste gang: Sendt rapport til Kyrre Enkel prototype Erik: Prorektor kommer, han må være med på noe fra 11:15. Møtes 10:30.

2016-04-18 Erik

Tilstede: Arnt, Sigve, Stian, Erik

Sensor: Aleksander Ballastvik - Vise fram at vi har gjort noe teknisk vanskelig, fått det til - Leverer en strøken rapport

To prototyper, stresser trafikkscriptet , interegrering mot Kyrre.

Om de to prototypene: Subjektivt mening, ta det litt med salt Hvorfor Fulgt en god utviklingsmetodikk Tom Røise! Git Visualisere git commit log

Sikre leveransen til Kyrre Sensor vil snakke med oppdragsgiver.

2016-04-18 Gruppe

Tilstede: Arnt, Sigve, Stian

Hva har blitt gjort sist uke Stian: Testing, consumer/producer testing. Python worker, queues, fredag fikset trafikkskript. Sigve: Testing, skrev om sikkerhet og litt testing, user manual. Monitorering m/Forskjellige løsninger. Arnt: Testing. Entrypoint (API, sende videre til queue, best practices) og DB utvikling (hente fra databasen, presenterer data)

Monitorering - Over til eget punkt/møte/whatever.

Toggl - Fortsette å bruke det. - Vær konsistens

2016-04-25 Erik

Tilstede: Arnt, Sigve, Erik

Ingenting spesielt

2016-04-25 Gruppe

Tilstede: Arnt, Sigve

What was was done last week? Sigve: Munin, Munin, Munin, og litt skriving. Arnt: Finishing database/entry. Begynne å skrive. Stian: Install worker on safe. Teste at durable queues overlever reboot. Implementerte retry connection på en av workererne. Satt opp bruk av yaml på alle workers. Left: Retry connection på resten. Unexpected sleep features. Litt code cleanup, samt se over all kommentering + evt pylint på workers/safe.

Toggl Looks good. A few hours here and there probbaly not counted for.

Tiden fremover/hva gjenstår: 3 uker. * 1 uker med Fabrikk * 2ish uker med skriving Stian: Retry connection på resten. Unexpected sleep features. Litt code cleanup, samt se over all kommentering + evt pylint på workers/safe. Sigve: A'ok! Mer skriving om sikkerhet rundt RabbitMQ og Flackr. Arnt: DogeMode feature! Brukerscript.

Monitor Munin, rabbitmq controlpanel. Stort sett a'ok!

Kyrre, testing 29/04 - Høre når han planlegger å ha det - Sigve planlegger å være der 29/04

(Dokumentasjon: Så mange figurer som mulig, skrive ferdig Flackr)

Møtet med Erik

2016-05-02 Gruppemøte

Tilstede: Arnt, Stian

Hva har blitt gjort Arnt: DogeMode, brukerscript, error handling (blant annet bildet), flackr slow query. Stian: Keepalive på worker mot queues, error handling, lint, skrive (zzz) Sigve: Skriving, monitor.

Trello Litt slack på Arnt og Stian.

Hva gjenstår - Flackr: Dele opp i versjoner. Branch/egne repo/tagging? - Flackr: Git commit på 192.168.200.130 til Flackr repo - Flackr: Skrive om replikering - Fabrikk: Passiv

Avslutningsmøte med Kyrre - Sende e-post i dag til Kyrre med ønske om møte. - Siste møte hvor vi planlegger overlevering

Passiv testing på fabrikk - sleep(randint(3, 5))

- Tag som kommer inn - Basert på tag så genereres en hvis tid

2016-05-02 Møte med Erik

Tilstede: Arnt, Stian...

Ingen Erik? Forsøkt å komme i kontakt med han uten svar.

EDIT 13:45: Han er i Trondheim! :D

2016-05-05 Skypemøte med Kyrre

- git repo med 3 versjoner - 2 forskjellige repoer -flackr - factory - Hver sin README som inneholder sin User Manual + configs (hele setup guiden) - Boookface README eksempel får vi på epost - Få noen til å teste det når vi er "ferdig" å sette sammen systemene - Levere REPO før 12 mai.

2016-05-09 Møte med gruppen

Tilstede: Arnt, Sigve, Stian

Trello - Litt slack, vær mer konsistens

Status skriving - Sigve: Monitorering starter i dag Sikkerhet done

- Stian: Generell requirements specification, startet med

- Arnt: Se på Flackr insert siden den ble endret

Oppsummering fra Kyrremøtet - Tagging - Legge til installasjonsguide i README.md.

Levering til Kyrre - Frigjøre en person til å jobbe med versjonsdelingen (ferdig mandag) | Arnt

- En person starter å se over Flackr (ferdig mandag) | Arnt

Møte med Erik - Spørre han pent om han kan lese igjennom en av røddagene

Eventuelt

———————————————— Rapport: Innledning Diskusjonspunktet (bra størrelse)

2016-05-09 Erik Hjelmås

- Erik leser igjennom Lørdag, så vi må ha det meste klart fredag kveld.

- Rapport: presis, vitenskapelige referanser? referanser: hvem, hva, hvor, når.

K Pre plan document

On the following page, the pre plan project report delivered in January is included.

BACHELOROPPGAVE:

Bookface 2.0

FORFATTERE:

Arnt-Helge Nilsen Øyan
Stian Svalstad
Sigve Næss

DATO:

28.01.2015

Sammendrag av Bacheloroppgaven

Tittel:	**Bookface 2.0**
Dato:	28.01.2015
Deltakere:	Arnt-Helge Nilsen Øyan Stian Svalstad Sigve Næss
Veiledere:	Erik Hjelmås
Oppdragsgiver:	Norwegian University of Science and Technology
Kontaktperson:	Kyrre Begnum, kyrre.begnum@hioa.no
Nøkkelord:	Norway, Norsk
Antall sider:	13
Antall vedlegg:	
Tilgjengelighet:	Åpen

Sammendrag: I Database- og Applikasjonsdrift har det blir brukt en tradisjonell tjenestearkitektur. I det siste har interessen for å bruke andre arkitekturer enn en tradisjonell LAMP stack. Dette skal vi se på i vår bacheloroppgave.

Summary of Graduate Project

Title:	**Bookface 2.0**
Date:	28.01.2015
Participants:	Arnt-Helge Nilsen Øyan Stian Svalstad Sigve Næss
Supervisor:	Erik Hjelmås
Employer:	Norwegian University of Science and Technology
Contact Person:	Kyrre Begnum, kyrre.begnum@hioa.no
Keywords:	SOA, Service Architecture Bookface2.0, Bachelor, IMT
Pages:	13
Attachments:	
Availability:	Open

Abstract: In Database- and application administration a traditional service architecture is used for Bookface. Lately there has been an increase in using different architecture than a LAMP stack. This is what we will look at in this bachelor thesis.

Contents

Contents . . . **iii**
List of Figures . . . **iv**
1 Introduction . . . **1**
1.1 Background . . . 1
1.2 Project goals . . . 1
1.3 Team members . . . 1
2 Extent . . . **2**
2.1 Task description . . . 2
2.2 Delimitation . . . 2
2.2.1 Concrete delimitation . . . 2
2.3 Field . . . 3
3 Project organization . . . **4**
3.1 Responsibilities and roles . . . 4
3.2 Procedures and code of conduct . . . 4
3.3 Tools . . . 4
3.3.1 ShareLaTeX . . . 4
3.3.2 Messenger . . . 4
3.3.3 Trello . . . 5
3.3.4 Git . . . 5
3.3.5 Google Docs . . . 5
4 Planning, supervision and report . . . **6**
4.1 Choice of system development model . . . 6
4.2 Plan for status meeting and decision point . . . 6
4.3 Required resources . . . 7
5 Organization of quality assurance . . . **8**
5.1 Risk Assessment . . . 8
5.1.1 Identifying and analyzing project risks . . . 8
5.1.2 Risk Analysis Table . . . 8
5.2 Documentation . . . 9
6 Plan for implementation . . . **10**
6.1 Milestones . . . 10
6.2 Gantt-chart . . . 10
Bibliography . . . **12**

List of Figures

1 Iterative development model 6
2 Gantt chart 11

1 Introduction

1.1 Background

In IMT3441 database and application administration at Norwegian University of Science and Technology you get introduced to one of the more traditional service architectures by implementing a web application called Bookface, a 1500th century micro blog. This is done by using the Linux Operating System, Apache HTTP Server, MySQL Database and PHP for web development. This is more commonly known as the LAMP stack. You as a student will be challenged to install, configure and operate this application as it gets hammered with traffic. Implementing load balancing between several web servers, caching technology on application level and more in order to maintain continuous operation.

Lately it has become more relevant to offer alternative architecture to benefit more from the course.

1.2 Project goals

Our current goals for this project is as follows

Learning

- Be able to understand and differentiate between different service architectures.
- Learn to install, configure and implement software solutions.
- Be able to design documentation that enhances quality of learning.
- Further develop knowledge of relevant professional areas like programming, network, scripting and database.
- Implement the use of "Best practices".

Performance

- Design two or more service-oriented architectures to compliment the current Bookface.
- Write documentation so students can configure, install and operate the service architectures.
- Develop a method to send traffic to the service architecture.

1.3 Team members

Our group consists of three persons from two different study programs. Arnt-Helge Nilsen Øyan and Stian Svalstad are studying Science in Network and System Administration (now IT Operations and Information Security). Sigve Næss is studying information security. All of us are third year students at NTNU in Gjøvik (former Høgskolen i Gjøvik).

2 Extent

2.1 Task description

IMT3441 database- and application operations has in recent years used a more traditional service architecture for its Bookface solution. Lately, it has been more relevant to provide an alternative architecture than the LAMP stack, so the students can benefit more from the course. Examples would be a node.js solution with API calls and unstructured databases, or a gaming application service. This will be in addition to Bookface, so the students can either choose or operate them all in this course.

The idea with this bachelor project is to design two or more alternative architectures that can be used in the course. Every architecture must consists of the following

- A codebase in a git repository, where the versions can tell a "story". For example you start off with a version with several mistakes or errors by the developers. Either they didn't have time or resources to fix this before it had to be deployed. Here you can use your creativity.
- Extra tools that are to be used from the "uptime" system, that directs traffic to the architecture and puts it under load, so that it feels more realistic. Much like the same way Bookface had users and posts added during the course.
- Documentation. It can't be too complex and preferably be based on standard packages in Ubuntu. One must also take in to account the technical expertise of the persons taking the course. If there is too much hacking, it might overshadow the big picture.

There is no equivalent course given today. This bachelor project will introduce an unique value for everyone who wants to learn more about administrating large scale systems. Furthermore, if the documentation is written in English, the possibility of offering this to international students in the future will be achievable. We will not be investigating the LAMP-stack as it is already in use in the current course. Our solution will have to be based on another stack.

2.2 Delimitation

As written in the task description, we are looking to design two or more alternative architectures. The extent of our ideas will decide whether two will be adequate or if more will be required. We will not be looking at Bookface or doing any means of development to it. Furthermore, we will not introduce configuration management in our architectures, as this is already covered in the course System Administration.

2.2.1 Concrete delimitation

Our employer has supplied some specifications that we have to work alongside.

- Final product has to be based on standard packages in Ubuntu
- Codebase in git repository, preferably with multiple versions.
- Documentation in English.

2.3 Field

This bachelor thesis will allow all of us to practice the skills we have learnt here at NTNU, and take them one step further into an unknown setting and applying them. This includes programming, scripting, database and system administration, operating systems and so forth. This project will challenge us to acquire new skills, such as learning a new programming language or an administrating a previously unknown architecture. Since one us has information security as his field, we will focus throughout the project to endeavour security by design.

3 Project organization

3.1 Responsibilities and roles

In our bachelor thesis we have chosen Arnt-Helge to be our project leader, and will therefore have some additional obligations. Our project leader will have the responsibility to maintain regular communication with our supervisor and employer. The project leader will also be responsible for scheduling additional meetings outside of the normal schedule if needed. Our employer for this bachelor thesis is Kyrre Begnum, which is the course coordinator for IMT3441 Database and Application Administration. In this course Bookface is used and maintained on the servers managed by the students. It is therefore very important for us to have good communication and regular meetings with Kyrre during the project period. Kyrre is also currently working as an Associate Professor at Oslo University College (HiOA). Also helping us we have Erik Hjelmås acting as our supervisor, which will assist us with technical competence and advice during the project. Erik is an Associate professor, Dr. scient and is currently the program director for IT operations and Information Security, formerly known as Science in Network and System Administration at NTNU. He also had a key role in development of the Norwegian Information Security Laboratory(NISlab).

3.2 Procedures and code of conduct

This project will require lots of collaboration, in order to deal with conflict situations and to ensure that the project is executed in an honorable fashion, a code of conduct (CoC) document has been developed within the team, which has been signed and agreed upon. It is available in appendix A.

3.3 Tools

For our project we have chosen 5 different tools which will help us with planning and organizing. All the tools have been reviewed by the group, and we have all agreed to use them on a regular basis throughout the project period.

3.3.1 ShareLaTeX

ShareLaTeX is an online LaTeX editor which allows for real tile collaboration and online compiling of projects to PDF format.[1] We decided to write our project in LaTeX, because it provides us with a high quality document compared to more traditional systems (e.g. GoogleDocs, Word, OpenOffice). Additionally NTNU have a LaTeX Bachelor Thesis template for us to use, which made it an easy choice. This tool also includes BibTeX for references. This will be used for its intended purpose.

3.3.2 Messenger

To communicate with the others in the project, we needed an easy application that was available for both Android and iOS. We decided to use Messenger, which is an instant messaging service and software application which provides text and voice communication.[2] It is also possible to quickly send files to the other group members while we are on the move.

3.3.3 Trello

To organize tasks, we have decided to use Trello. Trello is a collaboration tool that organizes your projects into boards. In one glance, Trello tells you what's being worked on, who's working on what, and where something is in a process.[3] We all have experience with using Trello in previous projects, and additionally it is available on all platforms (e.g. Android, iOS and Web) which makes it a perfect fit for us.

3.3.4 Git

The use of Git will provide us with a simple, fast and easy source code management system. Since we are going to hand in different versions of our system, using Git will make it easy for us to manage and organize them during the project period. Not to mention we can look at commits, changes and develop on branches. For this project we will be using Bitbucket by Atlassian [4].

3.3.5 Google Docs

Google Docs is a free web-based office suite, which allows users to create and edit documents online while collaborating with other users in real-time.[5] Since we already write our project in ShareLaTeX, the use of Google Docs will be minimal. But Google Docs is still a great and very easy accessible tool that we probably will use for smaller tasks like sharing notes and drafts.

4 Planning, supervision and report

4.1 Choice of system development model

We wanted to go with an agile model for this thesis. Being able to add, remove or change previous steps during design or development is a huge advantage. We also want some sort of product log since we might figure out features we want to implement during, but can't do it at the current time. The way IMT3441 works is you start by installing a webserver, put Bookface source files in the web root folder, install MySQL servers, and connect them together. It is an incremental way of working which we want to continue because Kyrre could possible use this in his class.

Depending on how our application will look, there will most likely be some development or scripting. This largely depends on the outcome of our ideas. Therefore we need our model to be flexible enough to handle the unpredictability.

Based on this and aspects earlier in the thesis we ended up with a incremental and iterative model [6]. There are several reasons for this. We have a product backlog we can use to base our work upon. This will be filled during the initialization step, where we will design our application architecture. We take something from our backlog, run it through an iteration phase. If everything goes well it can be added to our application. This is illustrated in figure 1.

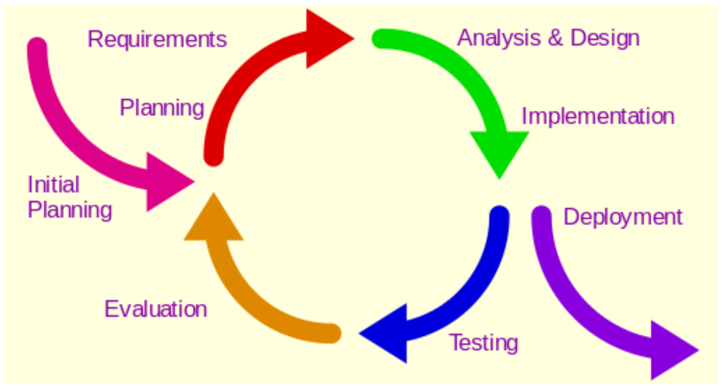


Figure 1: Iterative development model

4.2 Plan for status meeting and decision point

Each week we will have two planned meetings.

- Monday 10:15, group meeting
- Monday 13:30, meeting with supervisor, Erik Hjelmås.

Official meetings with our employer, Kyrre Begnum will for the most part predetermined since he is at NTNU only on fridays and is dependent on the train. Throughout the project we will have an open communication through e-mails, calls and Skype meetings if necessary so we do not have to wait for every friday.

4.3 Required resources

We need an environment where we can spin up virtual machines with Linux Operating System and network access. For this we will use the SkyHigh, an openstack implementation here at NTNU maintained by our supervisor. This is also the environment currently used by IMT3441s Bookface, which makes it perfect to test in.

5 Organization of quality assurance

5.1 Risk Assessment

5.1.1 Identifying and analyzing project risks

The chances that any unforeseen event that may affect the performance of the project while the project is in progress is real. To assess these, we use a categorization with 3 degrees within probability and 4 in consequence. For the **consequences** of an event, the different values represent the following scenario:

1. Slightly damaged — 2. Moderate damage — 3. Large damage — 4. Catastrophic damage

An event with little damage for the project will lead to between 1 and 3 day impact for implementation of the project within the time spent. Events with moderate damage for the project will result in up to 10 extra days for project implementation. Large damage of the will result in delays of up to 20 days. Catastrophic damage to the project involves more than 30 days delay or that the project can not continue.

For the **probability** of an event occurring, the following scale will be used:

1. Unlikely — 2. Probable — 3. Very likely — 4. Certainly

Unlikely means that the incident statistically will occur once per 5 years. A probable event will statistically occur once per year. An Event that is highly likely to occur will statistically occur once per six months. With a weighted model that uses these values it is possible to calculate which event must be prepared for, and which events that is not needed to prepare for.If the probability is low for an event occurring, the need of measures for the event will also be low. Any event with the weighted result over 5 must have measures.

5.1.2 Risk Analysis Table

Event	**Probability**	**Consequence**	**Weighted result**	**Measures**
Exceeding time frames	2	3	6	Yes
Break of contract between the group and the employer	1	4	4	No
The project is not documented well enough	2	3	6	Yes
Internal conflict in the group	2	3	6	Yes
Development/implementation of the system becomes too extensive	2	3	6	Yes
Long-term disease in group	1	4	4	No
Lack of expertise in group	1	3	3	No
Data loss	1	3	3	No

5.2 Documentation

Since the project is based on the purpose of teaching students, high quality and easily understandable documentation will be necessary. The documentation must be suitable for both the students, and the course leader. It is therefore very important that all documentation and notes during the project period is organized well so that all relevant information is presented in the final report. Our learnings and experiences should be strongly taken into consideration when preparing the final report. It will help us get a good understanding how we should present our findings for new students in an easy and efficient way. All newly added, changes or removals of any code/configuration will have to be documented. It is also strongly preferred that all relevant decisions we make is explained (for example why we chose X instead of Y, because of ...). Any code developed by us should be commented in a way so that it is easy understood by students that are new to the course.

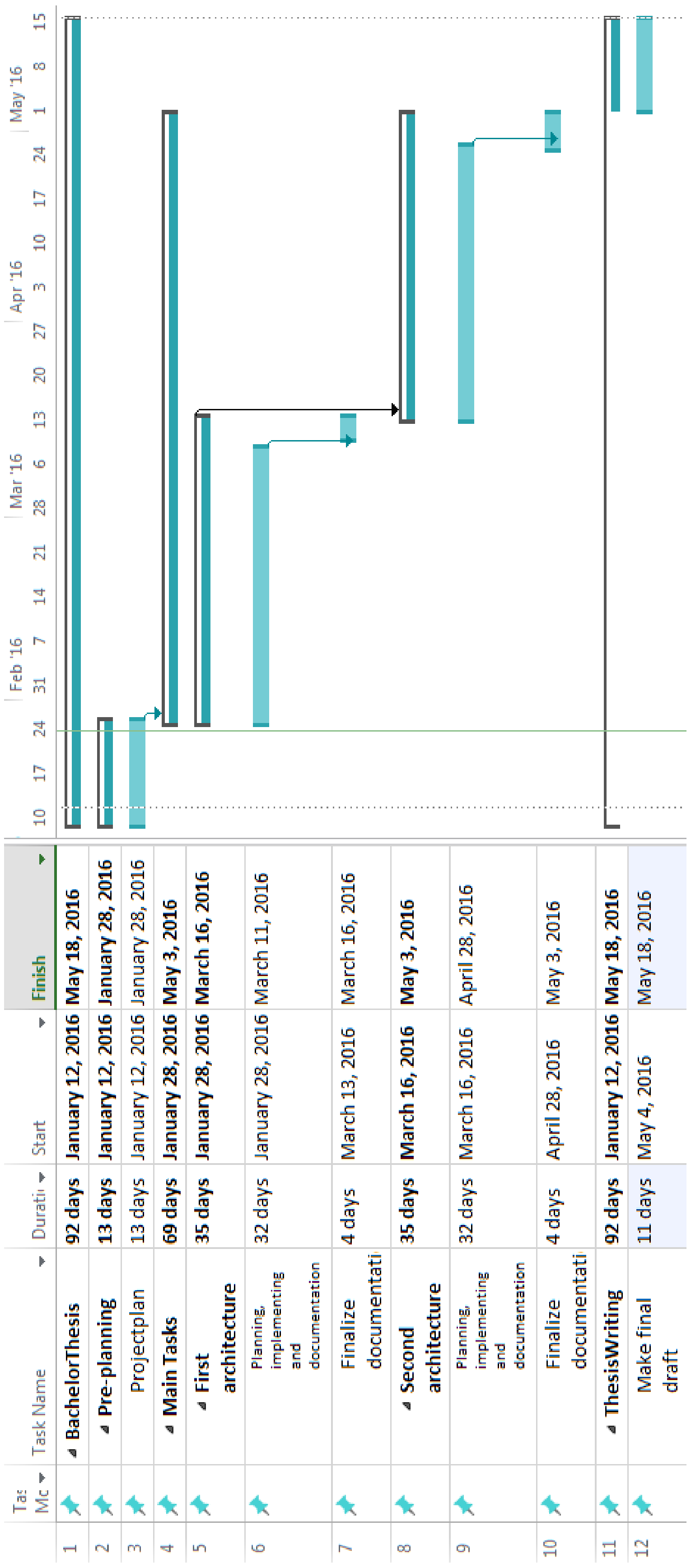
6 Plan for implementation

6.1 Milestones

- Pre-planning delivery date 28.01.2016
- Website. By end of February
- Final report delivery 18.05.2016

6.2 Gantt-chart

Our Gantt chart takes its characteristics from our project, as we are tasked with designing 2 architectures, we have split our development time into two sections. During this period of time we will take items off the product backlog and run it through the iteration phase. We decided to plan extra time for finalizing the documentation, this is due to the fact that it needs to be written in a way that enhances the learning experience. Lastly, we are finalizing the permanent draft of our own report for delivery.



	Tas Mc	Task Name	Durati	Start	Finish
1		**BachelorThesis**	**92 days**	**January 12, 2016**	**May 18, 2016**
2		**Pre-planning**	**13 days**	**January 12, 2016**	**January 28, 2016**
3		Projectplan	13 days	January 12, 2016	January 28, 2016
4		**Main Tasks**	**69 days**	**January 28, 2016**	**May 3, 2016**
5		**First architecture**	**35 days**	**January 28, 2016**	**March 16, 2016**
6		Planning, implementing and documentation	32 days	January 28, 2016	March 11, 2016
7		Finalize documentati	4 days	March 13, 2016	March 16, 2016
8		**Second architecture**	**35 days**	**March 16, 2016**	**May 3, 2016**
9		Planning, implementing and documentation	32 days	March 16, 2016	April 28, 2016
10		Finalize documentati	4 days	April 28, 2016	May 3, 2016
11		**ThesisWriting**	**92 days**	**January 12, 2016**	**May 18, 2016**
12		Make final draft	11 days	May 4, 2016	May 18, 2016

Figure 2: Gantt chart

Bibliography

[1] 2016. Sharelatex. https://www.sharelatex.com/learn/Learn:About/. (Visited Jan. 2016).

[2] 2016. Messenger. https://www.messenger.com/features/. (Visited Jan. 2016).

[3] 2016. Trello. http://help.trello.com/article/708-what-is-trello/. (Visited Jan. 2016).

[4] Atlassian. 2015. About bitbucket. [Online; accessed 22-January-2016]. URL: https://www.atlassian.com/software/bitbucket.

[5] 2016. Google docs. https://support.google.com/docs/answer/49008?hl=en/. (Visited Jan. 2016).

[6] Wikipedia. 2015. Iterative and incremental development — wikipedia, the free encyclopedia. [Online; accessed 18-January-2016]. URL: https://en.wikipedia.org/w/index.php?title=Iterative_and_incremental_development&oldid=680280418.

A Code of Conduct

1.Responsibilites

Arnt-Helge Nilsen Øyan will take the role as leader for this project. The project leader represents the group as a whole, and is considered to be the main point of contact.

2.Work Rules

Each member is expected to contribute with a minimum average of 25 hours per week. It is agreed upon that some weeks other subjects may require extra attention, due to exams or similar events. All members are expected to be on time for scheduled meetings. In case of absence, the project leader will have to be notified as soon as possible or preferably in advance. All members are expected to be at school monday-friday, unless otherwise agreed. Furthermore, we take different courses simultaneously and one member works one day per week, which means that all members will not be present at all times.

3.Decisions

Under any circumstances, when there are disagreements, voting will take place. If a majority of votes cannot achieved, external counseling will be required before a new voting session can take place. If there still are disagreements, at this stage the project leader will be given a double vote.

4.Costs

All costs that are related to this project, will be split evenly within the team.

5.Violations

Repeated violations to the CoC or lack of contribution, will lead to a meeting with the supervisor, and appropriate actions will be made.

[illegible]

Arnt-Helge Nilsen Øyan

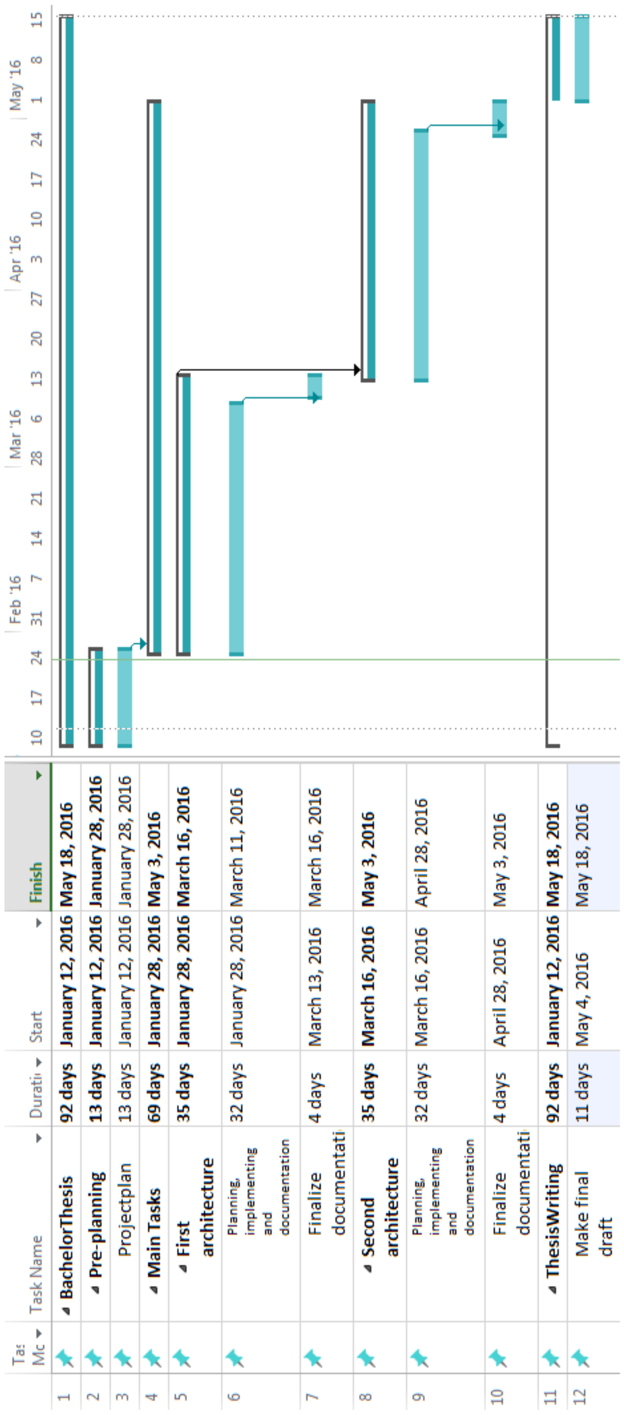
[illegible]

Stian Svalstad

[illegible]

Sigve Næss

L Gantt



	Ta: Mo	Task Name	Durati	Start	Finish
1		**BachelorThesis**	**92 days**	**January 12, 2016**	**May 18, 2016**
2		**Pre-planning**	**13 days**	**January 12, 2016**	**January 28, 2016**
3		Projectplan	13 days	January 12, 2016	January 28, 2016
4		**Main Tasks**	**69 days**	**January 28, 2016**	**May 3, 2016**
5		**First architecture**	**35 days**	**January 28, 2016**	**March 16, 2016**
6		Planning, implementing and documentation	32 days	January 28, 2016	March 11, 2016
7		Finalize documentati	4 days	March 13, 2016	March 16, 2016
8		**Second architecture**	**35 days**	**March 16, 2016**	**May 3, 2016**
9		Planning, implementing and documentation	32 days	March 16, 2016	April 28, 2016
10		Finalize documentati	4 days	April 28, 2016	May 3, 2016
11		**ThesisWriting**	**92 days**	**January 12, 2016**	**May 18, 2016**
12		Make final draft	11 days	May 4, 2016	May 18, 2016

Figure 6: Gantt chart