GRADEBADGE

DEVELOPMENT OF A CLOUD-BASED REWARD APPLICATION

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Erwin Toni Soekianto

April 2013

GRADEBADGE

DEVELOPMENT OF A CLOUD-BASED REWARD APPLICATION

A Project

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Erwin Toni Soekianto

April 2013

Approved by:

David Turner, Chair, Computer Science and Engineering — Date

Richard J. Botting

Arturo I. Concepcion

ABSTRACT

The purpose of this project is to investigate the use of cloud-based services to deliver cutting-edge applications. For this purpose, a prototype of a reward application using badges was developed to illustrate and explore this emerging paradigm. The cloud services utilized by this application include Heroku (for the application server), MongoLab (for the database), and Facebook (for authentication and social network integration). The application server is written in Javascript and runs inside a Nodejs execution environment. The application is accessed through a web browser running on either desktop or mobile computers.

On the client side, this application makes use of numerous web technologies, including HTML5, CSS, Bootstrap, and jQuery. The project also made use of the git version control system to manage source code and deployment of the application server to Heroku. The source code repository was stored remotely through the cloud-based service called GitHub.

The purpose of the GradeBadge application is to help organizations interact with and motivate their members in a fun way. It keeps the members engaged by giving badges as rewards for their efforts or achievements. In order to facilitate adoption amoung users, GradeBadge is integrated with the social networking site Facebook.

ACKNOWLEDGEMENTS

I would like to thank all the people with whom I have worked while pursuing my master's degree at California State University, San Bernardino (CSUSB). Studying in the School of Computer Science and Engineering at CSUSB has been a great learning experience. I would like to thank the faculty of the School of Computer Science and Engineering who supported this project by serving on my committee: Dr. David Turner, Dr. Arturo Concepcion and Dr. Richard Botting.

TABLE OF CONTENTS

Abstract iii

Acknowledgements iv

List of Tables ix

List of Figures x

1. Introduction 1

1.1 Background 1

1.2 Technology Overview 3

1.2.1 Facebook 3

1.2.2 Heroku 3

1.2.3 MongoDB 3

1.2.4 MongoLab 4

1.2.5 Git 4

1.2.6 Bootstrap 4

1.2.7 Jquery 4

1.2.8 Node.js 4

1.3 Project Purpose 5

1.4 Project Scope 5

1.5 Related Work 5

1.6 Definitions, Acronyms, and Abbreviations 6

2. Software Requirements Specification 9

2.1 External Interfaces Requirement 9

2.1.1 Hardware Interfaces 9

2.1.2 Software Interfaces 9

2.1.3 Communication Interfaces 9

2.2 Functional Requirement 10

2.2.1 Create group 10

2.2.2 Create Badge 11

2.2.3 Add Member 11

2.2.4 Issue Badges to Members 11

2.2.5 View Badge Earned 11

2.2.6 Share Badge to Social Networking 11

2.3 Performance Requirement 11

2.4 Design Constraint 11

2.5 Software System Attributes 12

3. System Architecture 13

3.1 Overview 13

3.2 Deployment Workflow 14

3.2.1 Developers/Release Manager 16

3.2.2 Heroku 16

3.2.3 MongoLab 17

3.2.4 Facebook 17

3.2.5 Client Browser 17

4. System Design 18

4.1 Design Overview 18

4.2 Model View Controller Architecture 18

4.3 Bandwidth Reduction Strategy 19
4.3.1 File Compression 19
4.3.2 Content Distribution Network Resource 20
4.3.3 Caching 20
4.4 Server-side Architecture Design 22
4.4.1 Configuration 23
4.4.2 Nodejs Module 24
4.5 Mapping of Model Classes to MongoDB 32
4.6 Request Handler Operation 36
4.7 Client-side Architecutre Design 39

5. Database Design 40
5.1 Overview 40
5.1.1 MongoDB 40
5.1.2 MongoLab 40
5.1.3 Documents 40
5.1.4 Collections 41
5.2 Data Model 41
5.2.1 User Collection 42
5.2.2 Group Collection 42
5.2.3 Badge Collection 42
5.2.4 Group Admin Links Collection 43
5.2.5 Group Member Links Collection 43
5.2.6 Badge User Links Collection 44
5.2.7 Group Badge Links Collection 44

6. Project Implementation 46
6.1 Loading Screen 47

6.2 Login Screen 48

6.3 Group Page 50

6.4 Add New Group Page 51

6.5 Group Page 52

7. Conclusion and Future Direction 53

7.1 Conclusion 53

7.2 Future Direction 53

APPENDIX A: Server Side Source Code 55

APPENDIX B: Model Classes Source Code 57

APPENDIX C: Request Handler Operation Source Code 60

APPENDIX D: Client Side Source Code 62

References 64

LIST OF TABLES

5.1 User Collection 42
5.2 Group Collection 42
5.3 Badge Collection 43
5.4 Group Admin Links Collection 43
5.5 Group Member Links Collection 44
5.6 Badge User Links Collection 44
5.7 Group Badge Links Collection 45

LIST OF FIGURES

2.1 Use Case Diagram 10

3.1 Deployment Diagram 14

3.2 System Integration 15

6.1 GradeBadge Loading Screen 47

6.2 GradeBadge Login Screen 48

6.3 Facebook Login Screen 49

6.4 GradeBadge Group Page 50

6.5 GradeBadge Add New Group Page 51

6.6 GradeBadge Group Page Added 52

1. INTRODUCTION

1.1 Background

A long time ago, businesses used to produce their own electric power. Due to an engineering breakthrough in electric generator and transmission methods, it became easier to produce and transmit electricity to businesses that once produced their own electricity. As more businesses started buying electric power, the production of electricity become less expensive, encouraging more compnies to purchase rather than make their own electricity.

And today, just like the utilities, instead of buying servers to run websites and applications, businesses rent servers and various services from cloud computing providers. These cloud resources are provided to companies in a way that resembles people renting apartments in a single building; even though you are in the same building with other people, you still have your own space. As more people rent and buy computing service from large-scale providers, the cost of these services are decreasing.

Today there are many cloud computing providers that provide different types of services. Some provide application hosting, databases, code repositories, authentication services, social network integration, etc. Some famous providers are Amazon, Google, Microsoft, Facebook, GitHub and Heroku.

Google App Engine provides infrastructure to build web applications on the same scalable systems that power Google applications. Google App Engine supports Python, Java and the Go programming languages. Google App Engine also provides several options for storing data, including App Engine Datastore, Google Cloud SQL, and

Google Cloud Storage. Windows Azure provides similar services as Google App Engine but supports a different set of programming languages, including ASP.NET, VB.NET, Java, Nodejs and Python.

Amazon also provides scalable cloud computing services, which includes the popular EC2 virtual machine instances, where users choose the type of operating system and configuration they need. The virtual machine service provided by Amazon is more flexible than language-specific executation environments such as Google App Engine and Heroku, but require more time and expertise to set up and manage.

In this project, Heroku is used as a cloud application platform for running Javascript in the Nodejs executation environment. Heroku also provide alternative execution environments that support Scala, Ruby, PHP and others. One of the benefits of using Heroku (and other similar application hosting services) is that the bandwidth and CPU capacity can be scaled up or down almost instantly to accomodate rapidly changing demand. Also, by relying on Heroku fto manage the hardware and system and network administration, developers gain the reliability, performance and security that is provided by a larger company with staff dedicated to these purposes.

Among all the programming language execution environments supported by Heroku, this project uses Nodejs, which supports server-side programming in Javascript. On the client side, the application uses HTML5, Javascript, Jquery library and the Bootstap framework. The applciation also relies on the MongoDB database as provided by the MongoLab service provider.

Node.js and MongoDB are often used together to build scalable web applications. The following describe the cloud computing service providers used in this project.

1.2 Technology Overview

1.2.1 Facebook

Facebook is a popular social networking website more than one billion users. It has proven to be a good platform to use for web applications that take advantage of its social networking features, such as friend lists, wall posting, etc [8]. Facebook also allows other applications to access user data with their authorization using its API.

1.2.2 Heroku

Application hosting is a form of cloud computing that enables developers to publish applications that require Internet connectivity. Heroku is one of the first companies to offer a remote Nodejs executation environment. Nodejs applications are deployed to Heroku using git [12]. To depoy, or redeploy, an applicaiton, the developer pushes a branch of a git repository to a remote git repository provided by Heroku. At deployment, Heroku extracts the application files from the git repository and runs the main executable, which is a server process. Heroku work is a partner of Facebook and so is designed to work easily with it.

1.2.3 MongoDB

There are many different types of cloud-based datastore services to choose from. For this project we used MongoDB because it works well with Node.js and Heroku. MongoDB is a scalable, high-performance, open source, NoSQL document-based database. MongoDB features include document-oriented storage, indexes, replication, high availability, auto-sharding, and querying [21].

1.2.4 MongoLab

MongoLab is the cloud computing provider for MongoDB database that was used in this project [22]. MongoLab has a free tier service that facilitates experimentation by developers, and thus was convenient for this project.

1.2.5 Git

Git is a distributed version control system [9]. This project uses GitHub to store and manage a remote git repository of application source code. Git was convenient for the project because Heroku uses git as a means to deploy web applications to its servers. Heroku's git-based deployment system allows easy creation of testing, staging, and production versions of the application.

1.2.6 Bootstrap

Bootstrap is a freely avaialble CSS and Javascript library created by Twitter. It provides a responsive design framework that works well for applications that run inside browsers in desktop computers, tablets and smart phones [4]. The application's user interface was contructed using Bootstrap.

1.2.7 Jquery

The GradeBadge application uses Jquery for AJAX transactions and DOM manipulation [19]. Bootstrap also depends on Jquery.

1.2.8 Node.js

Cloud-based services support apps written in several different programming languages, such as Java, Python, PHP, Javascript, Ruby and more. For this project we used Javascript running in a Node.js context. Node.js is a platform built on

Chrome's JavaScript runtime. It's purpose is to allow construction of fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient when used for I/O intensive applications such as Web applications [23].

1.3 Project Purpose

The purpose of this project is to explore the current technologies that enable rapid development and deployment of desktop and mobile Web applications that can scale to accomodate any number of users.

1.4 Project Scope

Project does not include database sharding features of MongoDB, which would allow a greater degree of scalability.

The GradeBadge application provides the following functionalities:

- Create Group
- Create Badge
- Add Member
- Issue Badges to Members
- View Badges Earned
- Share Badges with Social Networking Contacts

1.5 Related Work

There are other similar mobile Web application and cloud computing demonstration projects that were built as master's degree projects at CSUSB. One of these projects was completed by Manoj Kulkarni [32]. This project used Google App Engine for an application hosting provider and Google App Engine datastore and Java programming language. It also used Jquery Mobile as UI framework. The GradeBadge project uses different set of technologies, and is the first project at CSUSB that utilizes Nodejs and MongoDB. Most other similar projects at CSUSB have relied on either Java, .NET or PHP.

1.6 Definitions, Acronyms, and Abbreviations

The definitions, acronyms, and abbreviations used in the document are described in this section.

- GradeBadge: The name of this project.
- API: Application Programming Interface, which is a set of routines that an application uses to request and carry out low-level services performed by a computer's operating system; also, a set of calling conventions in programming that defines how a service is invoked through the application [10].
- Cloud computing: the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet) [**?**].
- JQuery: A javascript library for building web based applications [19].
- DOM: Document Object Model, which is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents [5]. item I/O: Input/Output.
- UI: User Interface.
- CSUSB: California State University, San Bernardino.

- HTML: HyperText Markup Language, which is the authoring language used to create documents on the Web [29].
- HTTPS: Hyper-text Transfer Protocol Secure, which is a secure network protocol used to encrypt data transferred between server and client [14].
- MVC: Model-View-Controller is an architectural pattern used in software engineering to isolate business logic from user interface considerations [33].
- UML: The Unified Modeling Language, which is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems [28].
- Microsoft Azure: Cloud computing platform provided by Microsoft [31].
- Google App Engine: Cloud computing platform provided by Google.
- Amazon Web Services: Cloud computing platform provided by Amazon [2].
- Heroku: Cloud application platform provided by Heroku [12].
- Android: Mobile operating system provided by Google [3].
- IOS: Mobile operating system provided by Apple [16].
- NoSQL: Types of databases that use key-value pairs for storing data unlike traditional relational databases. [24].
- JSON: Javascript Object Notation, which is a data representation format using key-value pairs [20].
- Ajax: Asynchronous JavaScript and XML, which is a method for Web applications to communicate between client browsers and servers [1].
- OOP: Object Oriented Programming, which is a computer programming paradigm where application logic is organized into objects that provide an interface to encapsulated state. [30].

- CDN: Content Distribution Network.

2. SOFTWARE REQUIREMENTS SPECIFICATION

2.1 External Interfaces Requirement

2.1.1 Hardware Interfaces

The application is hosted in the Heroku application cloud service. The Web server communicates over HTTPS to ensure that data transferred between client and server is untampered and private. The system is a Web based application; users are required to use a high-speed Internet connection and use an up-to-date Web browser.

2.1.2 Software Interfaces

JavaScript will be implemented throughout the website in order to display the correct feature the user requested. And HTML5 may be implemented throughout the website in order to display the correct feature the user requested.

2.1.3 Communication Interfaces

This application is designed to be viewed on any Internet Web browser provided that JavaScript and image features are enabled and the browser is HTML5 compatible. Performance may vary slightly between browsers; however, the functionality of the site should not be impaired.

2.2 Functional Requirement

The functions specified in this section directly correspond to work that will be conducted in this project as shown in the use case diagram in Figure 2.1.

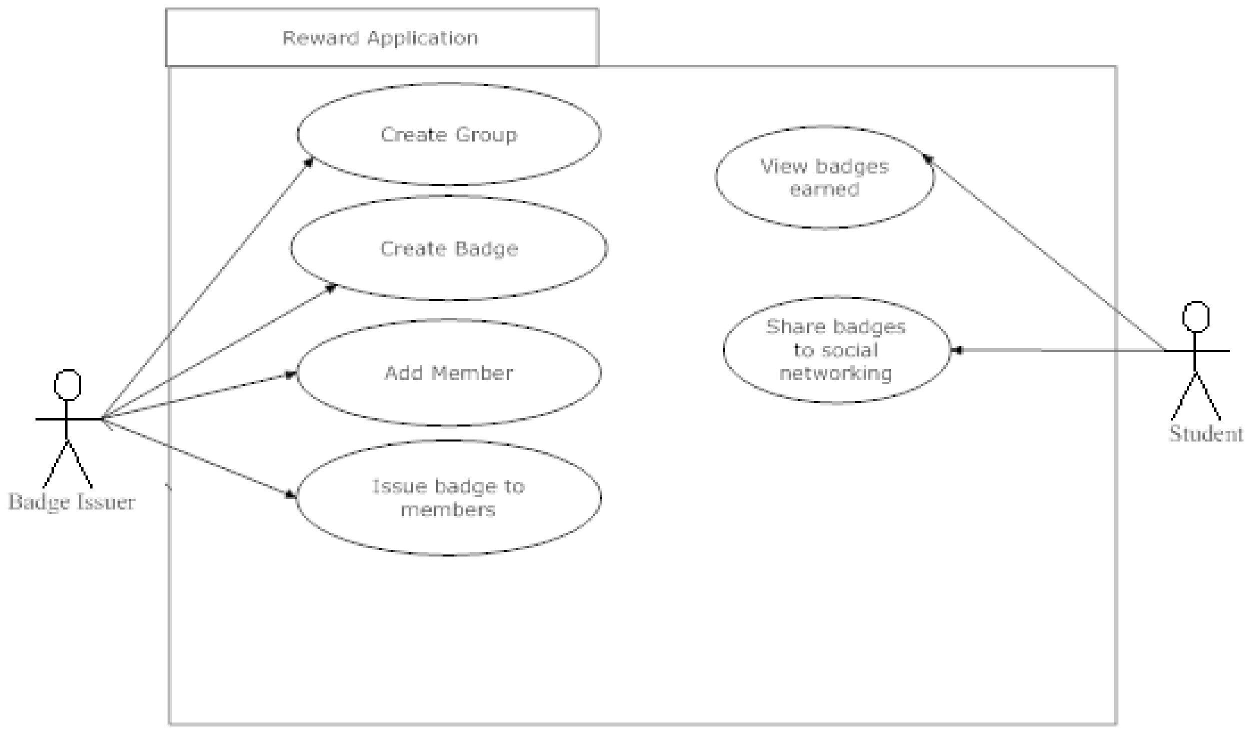


Fig. 2.1: Use Case Diagram

2.2.1 Create group

This functionality allows organizers or badge issuers to create groups, which will have a set badge collection.

2.2.2 Create Badge

This functionality allows badge issuers to create badge to be added to badge collection. A badge would have at least badge name and description.

2.2.3 Add Member

This functionality would allow organizers or badge issuers to add members into the groups as badge recipients. Member would have at least email address and name.

2.2.4 Issue Badges to Members

This functionality would allow badge issuers to issue badges to members.

2.2.5 View Badge Earned

This functionality would allow badge recipients to view all the badges that they have earned.

2.2.6 Share Badge to Social Networking

This functionality would allow badge recipients to share the badge to their social networking site such as Facebook

2.3 Performance Requirement

This application is going to be hosted in the Heroku cloud server, so the performance of this application would be high.

2.4 Design Constraint

This application requires internet-enabled devices and internet connection to perform. And every user must have Facebook account to be able to use this.

2.5 Software System Attributes

The author will keep coding standard with proper commenting and documentation.

3. SYSTEM ARCHITECTURE

3.1 Overview

There are three main server entities in this application, which are Heroku, Facebook, and MongoLab. When the client browser, either from desktop computer or tablet or smart phone, runs the application, it will hit the Heroku server, then the Heroku server will connect to Facebook server to check user authentication. And if the user is not logged-in, the user will be prompted with the Facebook login screen and submit the login data to Facebook server.

After the user is authenticated to use the application, the application server will connect to MongoLab server to read and write the data. The application server may occasionally go to Facebook server to check the users' friend list or post to user's wall or check whether the user is still logged-in.

This application uses HTTPS exclusively for security reasons, except in the local developer environment, where we use unencrypted HTTP, as shown in Figure 3.1.

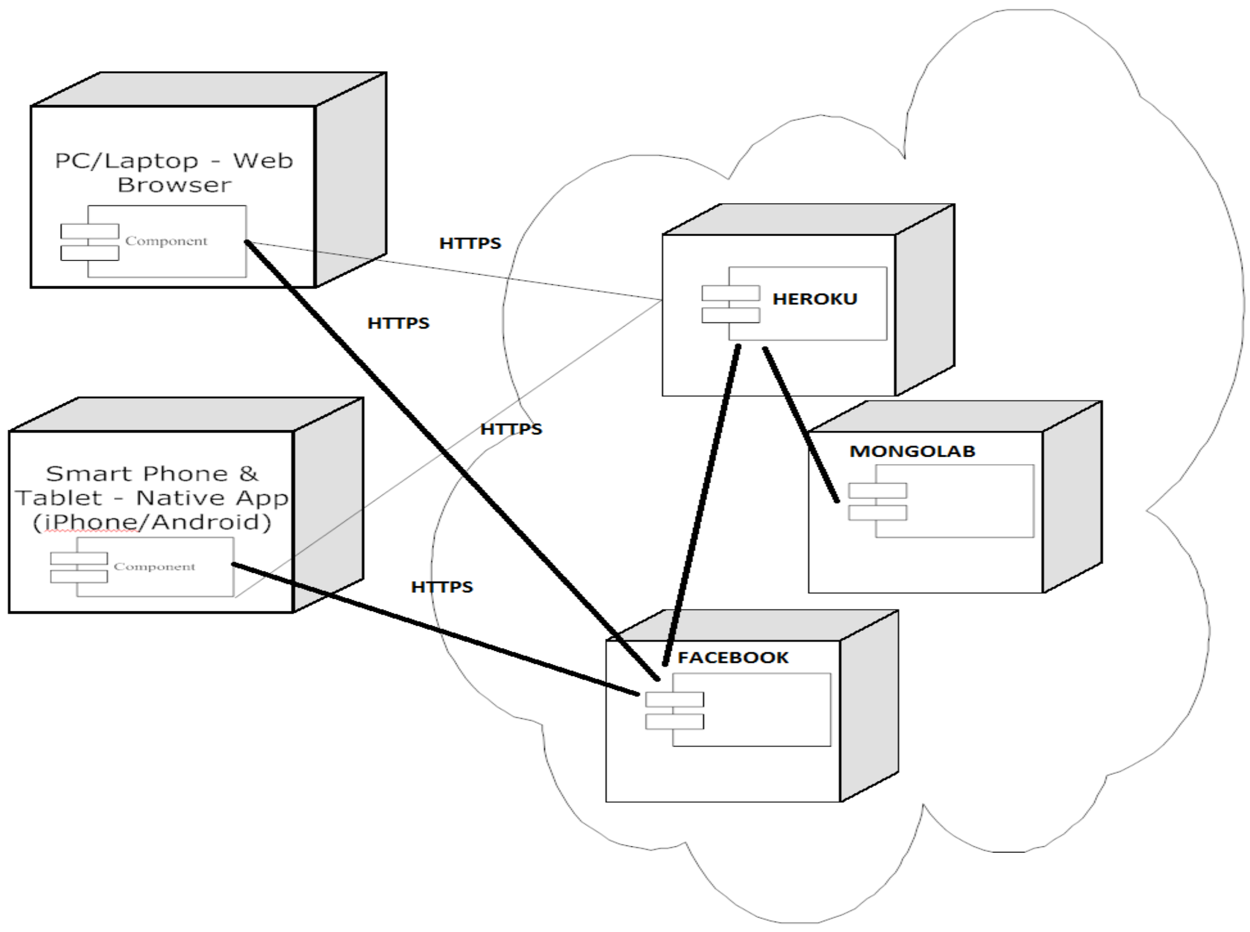


Fig. 3.1: Deployment Diagram

3.2 Deployment Workflow

There are three type of environments used in the deployment workflow: development, staging and production.

Developers work on new features or bug fixes in development branches then only minor updates are committed directly to the stable development branch. Once the features are implemented and/or set of bugs are fixed, they are merged in to staging branch and deployed to staging environment for testing and quality assurance. After testing is completed, the snapshop of staging branch is kept for production deployment, otherwise the process will repeat until the testing is completed. On the release date, the working staging branch is deployed to production environment.

Figure 3.2 illustrates the deployment workflow.

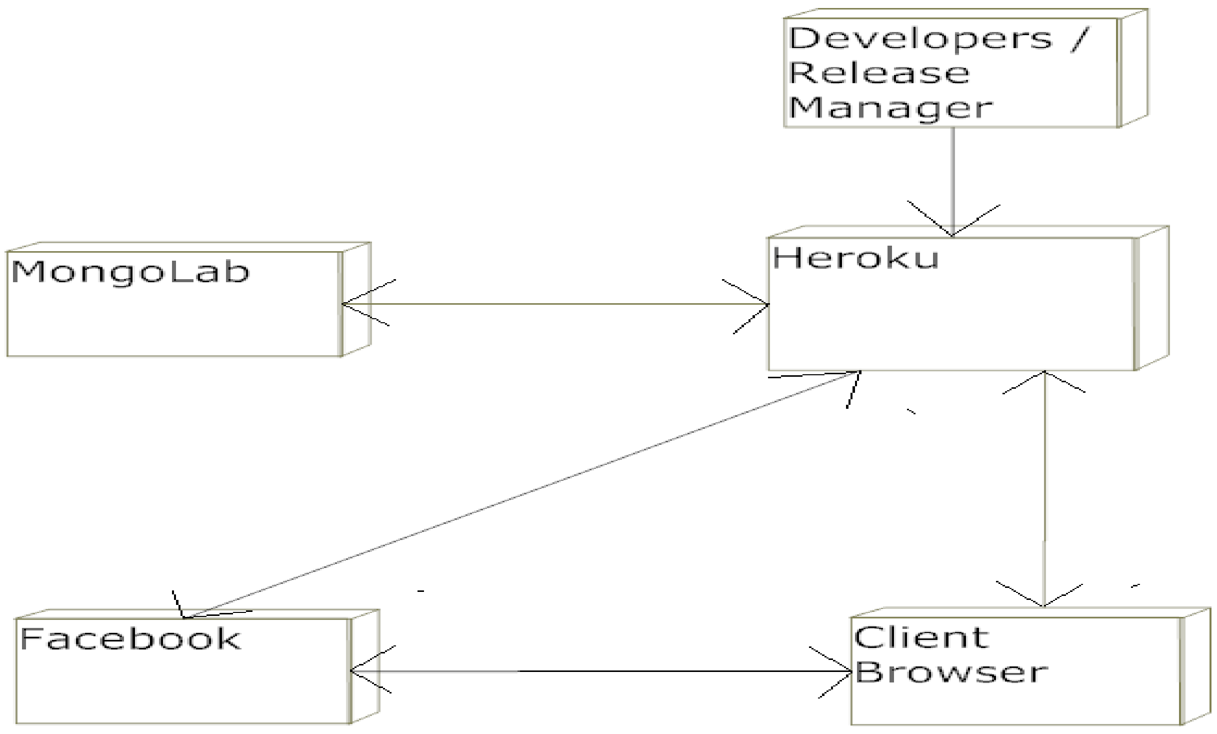


Fig. 3.2: System Integration

On this project, git is used as code repositories, to manage developments, staging and production branch. And Heroku toolbelt is also used to set the enviroment config variable for each deployment. Heroku allows users to use git to deploy automatically from local repositories.

3.2.1 Developers/Release Manager

In this project, developers first do unit testing in their local machines, then after the system reaches a certain point, the developer himself or assigned release manager will use git to push the changes to staging or production repositories that are connected to respective Heroku servers.

Below are the commands the developer/release manager uses to start the application in the staging enviroment.

```
$ foreman start
18:43:16 web.1   : started with pid 5540
18:43:16 web.1   : listening on 5000
```

Below are the commands for developer/release manager commits changes to master branch of the local repository, then followed by a push to master branch of the remote repository.

```
$ git status
$ git add .
$ git commit —m "message here"
$ git push origin master
```

Below are the commands for developer/release manager uses to push to staging environment.

```
$ git push staging master
```

3.2.2 Heroku

In this project there are two sets of heroku instances used: staging and production. The application running in Heroku connects to a database server running inside MongoLab using the MongoDB protocol to read and/or write data to database. Heroku also talks to Facebook servers via Facebook's Open Graph API.

Below is the sample command in Heroku to create staging environment

```
$ heroku create --remote staging
```

Below are the sample commands in Heroku to add environmetal variables in remote environment

```
$ heroku config:set S3_KEY=XXX --remote staging
$ heroku config:set S3_SECRET=YYY --remote staging
```

3.2.3 MongoLab

In this project there are three sets of mongo databases used: development, staging and production. It is important to keep the versions of databases since new versions of changes may include changes in database structure, so rolling back or forward the application version would not cause any error.

Below is the command on how to connect to remote Mongo Database that is hosted in MongoLab

```
$ mongo <servername>.mongolab.com:<portno>/<dbname>
    -u <dbuser> -p <dbpassword>
```

3.2.4 Facebook

Facebook is playing an important role in this project. Facebook provides user authentication and social media integration. Facebook allows connection using the Facebook API and Open Graph API.

3.2.5 Client Browser

Client browser uses HTTPS GET for static content, and HTTPS POST for AJAX request to Heroku. And client browser also connects to Facebook server directly using Facebook API and Open Graph API in HTTPS.

4. SYSTEM DESIGN

4.1 Design Overview

GradeBadge application uses a client-server architecture model, and it uses MVC (Model View Controller) architecute in the client and server side. And all incoming AJAX requests are submitted using HTTP POST and contain data encoded using JSON.

4.2 Model View Controller Architecture

This application is based on Model View Controller (MVC) Architecture. Model View Controller (MVC) architecture is a software design pattern for separating different components of a software application [33]. There are three main categories in the MVC architecture:

- Model : it represents data in the application. All the business rules are handled in the model.
- View: it represents UI components in the application. The UI components are responsible for presenting the model and for collecting user inputs.
- Controller: it is responsible for updating the data in the model and notifying the view about changes in the model.

4.3 Bandwidth Reduction Strategy

Following sections are the three strategies that are implemented to reduce and effectively use the bandwidth.

4.3.1 File Compression

In this application, in order to reduce the bandwidth from the server, files that are transferred from the application will be compressed in gzip format if the client browser supported it which will result to smaller file size. The following shows how this is done in code, when client requests file/s from server in req_file.js module, and the etag value is not matched.

```
if (file.gzip !== undefined &&
      req.headers['accept-encoding'] !== undefined &&
      req.headers['accept-encoding'].indexOf('gzip') !== -1) {
   return app_http.replyCached(res, file.gzip, file.type,
                                        file.etag, 'gzip');
} else {
   return app_http.replyCached(res, file.data,
                           file.type, file.etag);
}
```

Also the server will check if the files are not modified by comparing the etag value that will be part of caching strategy. The following shows how this is done in code, when the etag value is matched and server will not neccesary send back the requested file.

```
if (req.headers['if-none-match'] === file.etag) {
   return app_http.replyNotModified(res);
}
```

4.3.2 *Content Distribution Network Resource*

In this application, we are using Content Distribution Network (CDN) resources instead of storing those files in the server. So client browser will get it from the CDN rather than request them from the application server.

The following HTML element is used in the application to load the bootstrap CSS file from CDN.

```
<link href="//netdna.bootstrapcdn.com/twitter-bootstrap/
    2.2.2/css/bootstrap.css" rel="stylesheet">
```

4.3.3 *Caching*

In this application, we implemented the cache strategy where every request will be checked whether the file has been modified or is cached in the client browser. Etag value of the file is compared to decide whether the file has been modified or not. This strategy is implemented to reduce unneccesary bandwidth from the server

The following are the four possibilities of the respond from the server as HTTP header, the sample codes are taken from app_http.js file.

- Reply Not Found: This is the respond when the requested file is not found, and the server is going to return 404 Not Found to client, as shown in the following.

  ```
  res.writeHead(404, {});
  ```

- Reply Not Modified: This is the respond when the requested file is not modified, where the etag value is still the same. The following shows how this is done in the code, where it will just update the experation time of the requested file.

  ```
  res.writeHead(304, {
      'Connection'       : 'keep-alive',
      'Proxy-Connection' : 'keep-alive',
  ```

```
    'Cache-Control'      : 'max-age=31536000',
    'Expires'            : new Date(Date.now() + 31536000000)
  });
```

- Reply Not Cached: This is the respond when the requested file is not cached in the client browser. The following shows how this is done in the code, where server will respond by returning the requrested file/s.

```
res.writeHead(200, {
    'Content-Type'       : 'text/html',
    'Content-Length'     : buffer.length,
    'Connection'         : 'keep-alive',
    'Proxy-Connection'   : 'keep-alive',
    'Pragma'             : 'no-cache',
    'Cache-Control'      : 'no-cache, no-store'
  });
```

- Reply Cached: This is the respond when the requested file is cached in the client browser, then the server will update the expiration date of the file. The following shows how this is done in the code, where it checks if content-encoding is supported.

```
if(contentEncoding) {
    res.writeHead(200, {
      'Content-Type'       : contentType,
      'Content-Length'     : buffer.length,
      'Connection'         : 'keep-alive',
      'Proxy-Connection'   : 'keep-alive',
      'Pragma'             : 'public',
      'Cache-Control'     : 'max-age=31536000',
      'Vary'              : 'Accept-Encoding',
      'Expires'           : new Date(Date.now() + 31536000000),
      'ETag'              : etag,
      'Content-Encoding'  : contentEncoding
    });
  } else {
    res.writeHead(200, {
      'Content-Type'      : contentType,
      'Content-Length'    : buffer.length,
      'Connection'        : 'keep-alive',
      'Proxy-Connection'  : 'keep-alive',
      'Pragma'            : 'public',
      'Cache-Control'     : 'max-age=31536000',
      'Vary'              : 'Accept-Encoding',
      'Expires'           : new Date(Date.now() + 31536000000),
      'ETag'              : etag
    });
  }
```

4.4 Server-side Architecture Design

In Nodejs, components are organized in modules, in the form of namespace. All Nodejs modules that start with req_*.js are request handler that get requested from router.js. All ajax requests will go through req_op.js, that verifies the user logged-in to Facebook and app_version is current. Every req_op.js request must contains Facebook access_token and app_version. If the user is not logged-in to Facebook, req_op.js returns the following JSON document, login:true, if the version is not current, then req_op.js return the following JSON document, ver:true.

4.4.1 Configuration

Following are the list of configuration files required in the server side, and the source-code of these files can be found in the Appendix A.

- .env : This is the setup file that contains environment variables. This file only exist in developer local environment, and these values in this file would be set in each Heroku environment config for staging and production.

  Following are the environment variable values in development environment. FB_APP_ID is the Facebook application ID and FB_SECRET is the Facebook application secret, for each environment, we are using different Facebook application instance. PORT is the http port number set to use, and APP_VER is the current application version. MONGO_URI is the mongo database connection string, we are using different database insance for each environment.

  ```
  FB_APP_ID=466760923387961
  FB_SECRET=75e7a042473a989a3b876d3ec8749920
  PORT=5000
  APP_VER=2
  MONGO_URI=mongodb://app:gradebadge123@
            ds045757.mongolab.com:45757/gb-d
  ```

- .gitignore: This is the setup file that contains list of files or folders that will be ignored when committing or pushing to git repositories.

- .slugignore: This is the setup file that contains list of files or folders that will be ignored when calculating the slug limit in Heroku

- package.json: This is the setup file that contains of list of dependencies and engine version use in the application. This file also contains application name, version and description, as shown in the following.

```
{
    "name": "gradebadge",
    "version": "0.0.1",
    "description": "Grade Badge System",
    "dependencies": {
        "mongodb": "1.2.13"
    },
    "engines": {
        "node": "0.8.21",
        "npm": "1.2.12"
    }
}
```

- Procfile: This is the setup file that tells Heroku how to launch the application, as shown in the following.

```
web: node main.js
```

4.4.2 Nodejs Module

The following list below are the list of files that are used in server executions, and the source-code of these files can be found in the Appendix A.

- main.js: This is the main module in node js, which contains the code that verify all neccessary environment variables are set correctly. It also invoke initialization in neccesarry modules to start the application, after the initialization is completed, it starts the HTTP request handling loop.

Following is the part of code in the main.js module, where it initializes the model, router and fb module and run them asynchronously.

```
var n = 3;
function done() {
  if (--n === 0) {
    router.start();
  }
}
model  .init(done);
router .init(done);
fb     .init(done);
```

- router.js: This module routes incoming requests to the right module. Following is the part of the code in router.js module where it checks for pathname of incoming request and routes them to corresponding module handler.

```
function route(req, res) {
  var pathname = url.parse(req.url).pathname;
  if      (pathname === '/') req_root.handle(req, res)
  else if (pathname === verpath) req_app.handle(req, res)
  else if (pathname === issuerpath) req_issuer.handle(req, res)
  else if (pathname === '/mem')  req_mem.handle(req, res)
  else if (pathname === '/counters') req_counters.handle(req, res)
  else req_rootdir .handle(req, res);
}
```

- app_ajax.js: This module contains the application wide AJAX handling routines. Following is the sample function of Ajax respond that would send data back to client in JSON and utf8 format.

```
exports.data = function(res, data) {
  if (data === undefined) {
    data = {};
  }
  var buf = new Buffer(JSON.stringify({'data' : data}), 'utf8');
  res.writeHead(200, {
    'Content-Type': 'application/json; charset=UTF-8',
    'Content-Length': buf.length,
    'Pragma': 'no-cache',
    'Cache-Control': 'no-cache, no-store'
  });
  res.end(buf);
};
```

- app_http.js: This module contains all of HTTP protocol routines for the application, caching headers, compression header and other HTTP based optimization are implemented in this module. This file was discussed in the caching strategy section.

- fb.js: This module contains all code that interact with Facebook. Following is the sample of initialization function, that will check Facebook App ID and secret below.

```
exports.init = function(cb) {
  var options = {
    hostname: 'graph.facebook.com',
    path: '/oauth/access_token?' +
          'client_id=' + process.env.FB_APP_ID +
          '&client_secret=' + process.env.FB_SECRET +
        '&grant_type=client_credentials',
    method: 'GET'
  };
  send(options, function(data) {
    if (data instanceof Error) {
      throw data;
    }
    if (data.access_token === undefined) {
      throw new Error(
        'fb.init: access_token not returned by facebook.' +
        '\nfb.init: Facebook returned: ' + JSON.stringify(data)
      );
    }
    appToken = data.access_token;
    cb(appToken);
  });
};
```

- logger.js: This module contains application wide logging functionalities. Following is the sample of error logging function in this module.

```
exports.errors = function(msg, opt_msg) {
  if (errorsPrinted < process.env.LOGGER_MAX_ERRORS) {
    ++errorsPrinted;
    print('ERROR', msg, opt_msg);
    if (errorPrinted == process.env.LOGGER_MAX_ERROR) {
      console.log('MAX ERROR HIT');
    }
  }
  ++exports.errorsReceived;
};
```

- model.js: This module initializes the database connnection pool during server start. Following is the sample code to connect to Mongo client using the MONGO_URI value in the environment variable, and connection options that were set in the model module.

```
MongoClient.connect(process.env.MONGO_URI, connectOptions, function(
        if (err) throw err;
        exports.db = db;
        cb();
    });
```

- req_app.js: This module handles request for application HTML template for badge earner. Following is the initialization function of req_app.js module, where it return app.html template, replace the FB_APP_ID with the one is environment variable, set the etag value and compress the file to gzip format.

```
exports.init = function(cb) {
    fs.readFile('app.html', 'utf8', function(err, file) {
        if (err) throw err;
        html = new Buffer(file.replace(/FB_APP_ID/g,
            process.env.FB_APP_ID), 'utf8');
        etag = app_http.etag(html);
        zlib.gzip(html, function(err, result) {
            if (err) throw err;
            ghtml = result;
            cb();
        });
    });
};
```

- req_counter.js: This module handles request for logging counter. The following is the request handler function of req_counter module where it contruct the page with logging information and send it back to client in utf8 format and not cached.

```
exports.handle = function(req, res) {
    var page =
                '<p>logger errors: ' + logger.errorsReceived
+ '</p>' +
                '<p>logger warnings: ' + logger.warningsReceived + '</p
                '<p>logger info: '  + logger.infoReceived
+ '</p>' +
                '<p></p>';
        page = new Buffer(page, 'utf8');
    app_http.replyNotCached(res, page);
}
```

- req_file.js: This module handles request for static content. The request handler function of this module is discussed in Bandwidth Reduction Strategy chapter where it handles the etag and gzip file compression.

  The following is the sample function where it calculates and displays memory consmption of the server to the console.

```
function displayStats(files) {
    var uncompressed = 0, compressed = 0;
```

```
  for (var i = 0; i < files.length; ++i) {
    uncompressed += files[i].data.length;
    if (files[i].gzip !== undefined) compressed += files[i].gzip.len
  }
  console.log('memfile bytes, uncompressed: ' +
         Math.ceil(uncompressed / 1024 / 1024) + ' MB');
  console.log('memfile bytes, compressed:   ' +
         Math.ceil(compressed / 1024 / 1024) + ' MB');
}
```

- req_issuer.js: This module handles request for application HTML template for badge issuers. Following is the initialization function of req_issuer.js module, where it return issuer.html template, replace the FB_APP_ID with the one is environment variable, set the etag value and compress the file to gzip format.

```
exports.init = function(cb) {
  fs.readFile('issuer.html', 'utf8', function(err, file) {
    if (err) throw err;
    html = new Buffer(file.replace(/FB_APP_ID/g,
          process.env.FB_APP_ID), 'utf8');
    etag = app_http.etag(html);
    zlib.gzip(html, function(err, result) {
      if (err) throw err;
      ghtml = result;
      cb();
    });
  });
};
```

- req_mem.js: This module handles request for memory usage. The following is the request handler function of req_mem module where it contruct the page with memory usage information and send it back to client in utf8 format and not cached.

```
exports.handle = function(req, res) {
  var usage = process.memoryUsage(),
      page = '<p>Heroku limit = 512 MB</p>' +
             '<p>rss = '       + Math.ceil(usage.rss
/ 1024 / 1024) + ' MB</p>' +
             '<p>heapTotal = ' + Math.ceil(usage.heapTotal / 1024 /
             '<p>heapUsed = '  + Math.ceil(usage.heapUsed
/ 1024 / 1024) + ' MB</p>';
      page = new Buffer(page, 'utf8');
  app_http.replyNotCached(res, page);
}
```

- req_root.js: This module handles request for static content under the root URL. The following is the request handler function of req_root module where it contruct the html page that will redirect the page to the correct path with current version number.

```
var html = new Buffer('<script>location.replace("/' + process.env.AP
exports.handle = function(req, res) {
  app_http.replyNotCached(res, html);
};
```

- req_op.js : This module handles all AJAX request from client and routes to appropriate modules. The following is the part of request handler function where it checks the pathname of incoming request and call the apporiate module handler.

```
exports.handle = function(req, res) {
  app_ajax.parse(req, function(data) {
      var pathname = url.parse(req.url).pathname;
      if (pathname === '/op/save-group') {
        op_save_group.handle(data, res);
      }else if (pathname === '/op/read-groups-by-admin') {
        op_read_groups_by_admin.handle(data, res);
    });
  });
}
```

4.5 Mapping of Model Classes to MongoDB

There will be one node js module to represent the mongoDB collection, named model_(collection_name).js. And many-to-many relationships are represented by linking documents, named (a)_(b)_links

The following list below are the list of files that are used to Model to represent MongoDB, and the source-code of these files can be found in the Appendix B.

- model_group.js: this node.js module represents Groups collection. Following is one of functions in the model_group to get group document by given id.

```
exports.getByIds = function(group_ids, cb){
  model.db.collection('groups').find({'_id' : {$in: group_ids} }).to
    model.db.close();
    if (err) return cb(err);
    cb(groups);
  });
};
```

- model_badge.js: this node.js module represents Badges colletion. Following is one of functions in the model_badge module to create badge document in the badge collection from given document.

```
exports.create = function(badge, cb) {
  model.db.collection('badges').insert(
    badge,
    function(err) {
      model.db.close();
      if (err) return cb(err);
      cb();
    }
  );
};
```

- model_user.js: this node.js module represents Users collection. Following is one the functions in the model_user module to record the login activity of the user by updating the last login time stamp in the user document.

```
exports.login = function(user, cb) {
  model.db.collection('users').save(
    user,
    function(err) {
      model.db.close();
      if (err) return cb(err);
      cb();
    }
  );
};
```

- model_group_admin.js: this node.js module represents group_admin_links collection. Following is one of the function in the model_group_admin module to get the array of groups ids from given admin (user) id.

```
exports.getGroupIdsByAdminId = function(admin, cb) {
  model.db.collection('group_admin_links',{'gid' : true}).find(admin
    model.db.close();
    if (err) return cb(err);
    var group_ids = group_admin_links.map(function(group_admin_link)
    cb(group_ids);
  });
};
```

- model_group_member.js: this node.js module represents group_member_links collection. Following is one of the function in the model_group_member module to get the array member (user) ids from given group id.

```
exports.getMemberIdsByGroupId = function(group, cb) {
  model.db.collection('group_admin_links',{'uid' : true}).find(group
    model.db.close();
    if (err) return cb(err);
    console.log('model_group_admin getMemberIdsByGroupId
group_admin_links = '+ JSON.stringify(group_admin_links));
    var member_ids = group_admin_links.map(function(group_admin_link
    cb(member_ids);
  });
};
```

- model_user_badge.js: this node.js module represents user_badge_links collection. Following is one of the functions in model_user_badge module to create the link document.

```
exports.create = function(user_badge, cb) {
  model.db.collection('user_badge_links').insert(
    user_badge,
    function(err) {
      model.db.close();
      if (err) return cb(err);
      cb();
    }
  );
};
```

- model_group_badge.js: this node.js module represents group_badge_links collection. Following is one of the functions in model_group_badge module to create the link document.

```
exports.create = function(group_badge, cb) {
  model.db.collection('group_badge_links').insert(
    group_badge,
    function(err) {
      model.db.close();
      if (err) return cb(err);
      cb();
    }
  );
};
```

There will be one node js module to handle ajax request from client, named op_(request).js. Every request may read, write or update to and from more than one collection

The following list below are the list of files that are used in request handler operation, and the source-code of these files can be found in the Appendix C

- op_read_badges_by_group.js: this operation is for request for all badges in the given group. The following shows how this is done in the code, where it gets the array of badge ids with given group id from group_badge links document, and after that get the badges document based on the badge ids array result from badge collection.

```
exports.handle = function (data, res) {
  var group = { gid: data.gid };
  model_group_badge.getBadgeIdsByGroupId(group, function(bids) {
    if (bids instanceof Error) {
      return app_ajax.error(res);
    }
    model_badge.getByIds(bids, function(badges){
      if (badges instanceof Error) {
        return app_ajax.error(res);
      }
      return app_ajax.data(res, badges);
    });
  });
};
```

- op_read_groups_by_admin.js: this operation is for request for all groups in the given admin. The following shows how this is done in the code, where it gets the array of group ids with given admin id from group_admin links document, and after that get the groups document based on the group ids array result from group collection.

```
exports.handle = function (data, res) {
  var admin = { uid: data.uid };
  model_group_admin.getGroupIdsByAdminId(admin, function(gids) {
    if (gids instanceof Error) {
      return app_ajax.error(res);
    }
    model_group.getByIds(gids, function(groups){
      if (groups instanceof Error) {
        return app_ajax.error(res);
      }
      return app_ajax.data(res, groups);
    });
  });
};
```

- op_save_badge.js: this operation is for request for saving given badge details. The following shows how this is done in the code, where it adds badge document to badge collection, and get the newly created badge id together with group id then add to group_badge linking document.

```
exports.handle = function (data, res) {
  var badge = { name: data.name, desc: data.desc, pict:data.pict, gi
  model_badge.create(badge, function(err) {
    if (err) {
      return app_ajax.error(res);
    }
    var group_badge = { bid : badge._id , gid : badge.gid };
    model_group_badge.create(group_badge , function(err) {
    if (err) {
      return app_ajax.error(res);
    }
    });
  });
  return app_ajax.data(res , {gid : badge._id} );
};
```

- op_save_group.js: this operation is for request for saving given group details. The following shows how this is done in the code, where it adds group document to group collection, and get the newly created group id together with user id then add to group_admin linking document.

```
exports.handle = function (data, res) {
  var group = { name: data.name, desc: data.desc , uid: data.uid };
  model_group.create(group , function(err) {
    if (err) {
      return app_ajax.error(res);
    }
    var group_admin = { gid : group._id , uid : group.uid };
    model_group_admin.create(group_admin , function(err) {
      if (err) {
        return app_ajax.error(res);
      }
    });
```

```
        return app_ajax.data(res, {gid : group._id} );
    });
};
```

4.7 Client-side Architecutre Design

The following list below are the list of files that are used in client-side execution, and the source-code of these files can be found in the Appendix D.

- app.html : This is the html template for badge earner page.
- issuer.html : This is the html template for badge issuer page.
- public_root/channel.html : This is the static content required by Facebook.
- public_root/favicon.ico : This is the statuc content for icon use in the browser.
- public_ver/app.js : This is the client side java-script.
- public_ver/style.css: This is the css file used in this application.

5. DATABASE DESIGN

5.1 Overview

5.1.1 MongoDB

MongoDB is a scalable, high-performance, open source, NoSQL document-based database. MongoDB features include document-oriented storage, indexes, replication, high availability, auto-sharding, and querying. [21]

5.1.2 MongoLab

MongoLab is the cloud computing provider for MongoDB database that was used in this project [22]. MongoLab has a free tier service that facilitates experimentation by developers, and thus was convenient for this project.

5.1.3 Documents

Data in MongoDB is stored in documents, and every document must have a primary key named _id. Like reqular SQL-based database, documents are like row in a table, where in MongoDB, documents have flexible schema, but every document must have _id field, even if you don't speficy the _id field, mongoDB will add that automatically. [21]

Although document structure is not enforced in MongoDB, different data model and structure may have significant impacts on MongoDB and application performace. So it is good to keep some kind of structure or pattern in data model.

5.1.4 Collections

Documents in MongoDB are organized in collection, and basic database operations are performed based on collection. Indexes can be assigned in any field or subfield contained in documents within a MongoDB collection, and they are defined on per-collection level. [21]

5.2 Data Model

In this project, there are three main collections, User, Group and Badge Colletion. And there are four linkings colletions to represent the many-to-many relationship between the main collections.

- user : this collection contains the user information
- group : this collection contains the group information
- badge : this collection contains the badge information
- group_admin_links : this linking collection contains group_id and user_id, which represent the admin of a group
- group_member_links : this linking collection contains group_id and user_id, which represent the member of a group
- badge_user_links : this linking colletion contains badge_id and user_id, which reperesnt badges that user earned
- group_badge_links : think linking collection contains group_id and badge_id, which represent badges that belong to a group

5.2.1 User Collection

This collection contains of user documents, contain the following information, Table 5.1

Tab. 5.1: User Collection

Document Attribute	Sample Data	Description
_id	34728373	Facebook user ID
last_login	12/1/13 12:34:56 PM	User last login timestamp

5.2.2 Group Collection

This collection contains of group documents, Table 5.2

Tab. 5.2: Group Collection

Document Attribute	Sample Data	Description
_id	51466591b95b2b5023000001	Auto-generated Group ID
name	Yoga CSUSB	Group name
desc	This is Yoga group in CSUSB	Group description

5.2.3 Badge Collection

This collection contains of badge documents, Table 5.3

Tab. 5.3: Badge Collection

Document Attribute	Sample Data	Description
_id	514e34507075ae0c0f000001	Auto-generated Badge ID
name	Attended 10 classes	Badge name
desc	This is badge description	Badge description
pict	pict1.png	Badge Picture Reference

5.2.4 Group Admin Links Collection

This collection represent many-to-many relationship between group and user. User in this collection means the admin of the group. Table 5.4

Tab. 5.4: Group Admin Links Collection

Document Attribute	Sample Data	Description
_id	514e34507075ae0c0f000001	Auto-generated Document ID
gid	514e34507075ae0c0f000123	Group ID
uid	514e34507075ae0c0f000456	User ID as Admin

5.2.5 Group Member Links Collection

This collection represent many-to-many relationship between group and user. User in this collection means the member of the group. Table 5.5

Tab. 5.5: Group Member Links Collection

Document Attribute	Sample Data	Description
_id	514e34507075ae0c0f000002	Auto-generated Document ID
gid	514e34507075ae0c0f000123	Group ID
uid	514e34507075ae0c0f000675	User ID as Member

5.2.6 Badge User Links Collection

This collection represent many-to-many relationship between badge and user. User in this collection means the user who earned the badge. Table 5.6

Tab. 5.6: Badge User Links Collection

Document Attribute	Sample Data	Description
_id	514e34507075ae0c0f000005	Auto-generated Document ID
bid	514e34507075ae0c0f000975	Badge ID
uid	514e34507075ae0c0f000456	User ID as Earner

5.2.7 Group Badge Links Collection

This collection represent many-to-many relationship between group and badge. Badge in this collection means the badge in the group. Table 5.7

Tab. 5.7: Group Badge Links Collection

Document Attribute	Sample Data	Description
_id	514e34507075ae0c0f000004	Auto-generated Document ID
gid	514e34507075ae0c0f000158	Group ID
bid	514e34507075ae0c0f000953	Badge ID

6. PROJECT IMPLEMENTATION

The GradeBadge application is designed to work on mobile, tablet devices and desktop computers. The UI of the application is developed using Bootstrap. When a page requires the data to be loaded from server or modified or deleted, a request is sent to the Web server over HTTPS. The requests are sent to the Web server using Ajax. For handling Ajax requests and responses, this application uses JQuery Ajax API. All UI components are dynamically created or initialized in response to the data received from the Web server.

6.1 Loading Screen

When the GradeBoard application is loaded, a loading screen is presented to the user as shown in the Figure 6.1. The loading screen shows application logo and loading progess bar, and the screen is automatically redirected after the loading completed.

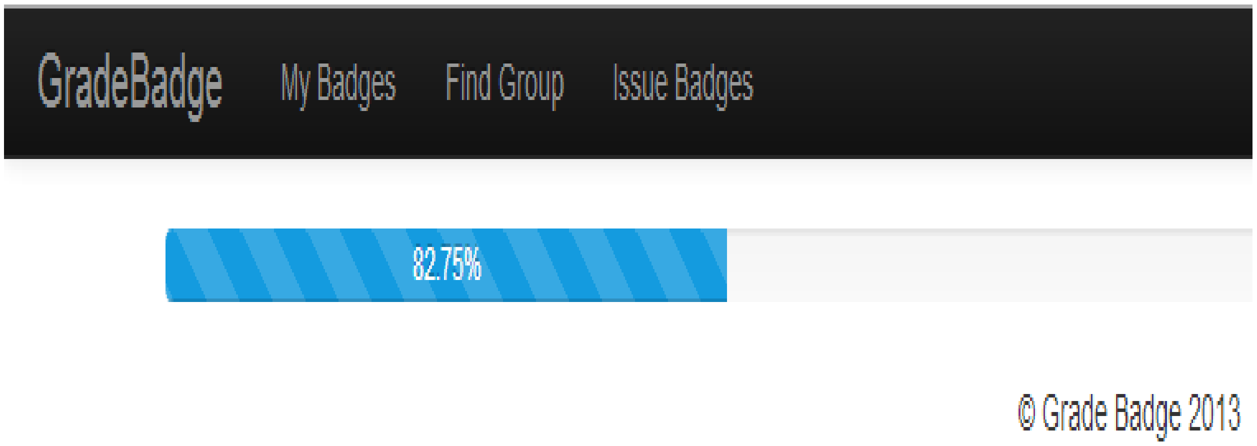


Fig. 6.1: GradeBadge Loading Screen

6.2 Login Screen

GradeBadge uses Facebook account for users to login. When the user is not logged-in to Facebook, the screen is automatically redirected to the Facebook login screen as shown in Figure 6.2. Every user in the system can be badge issuer and badge earner.

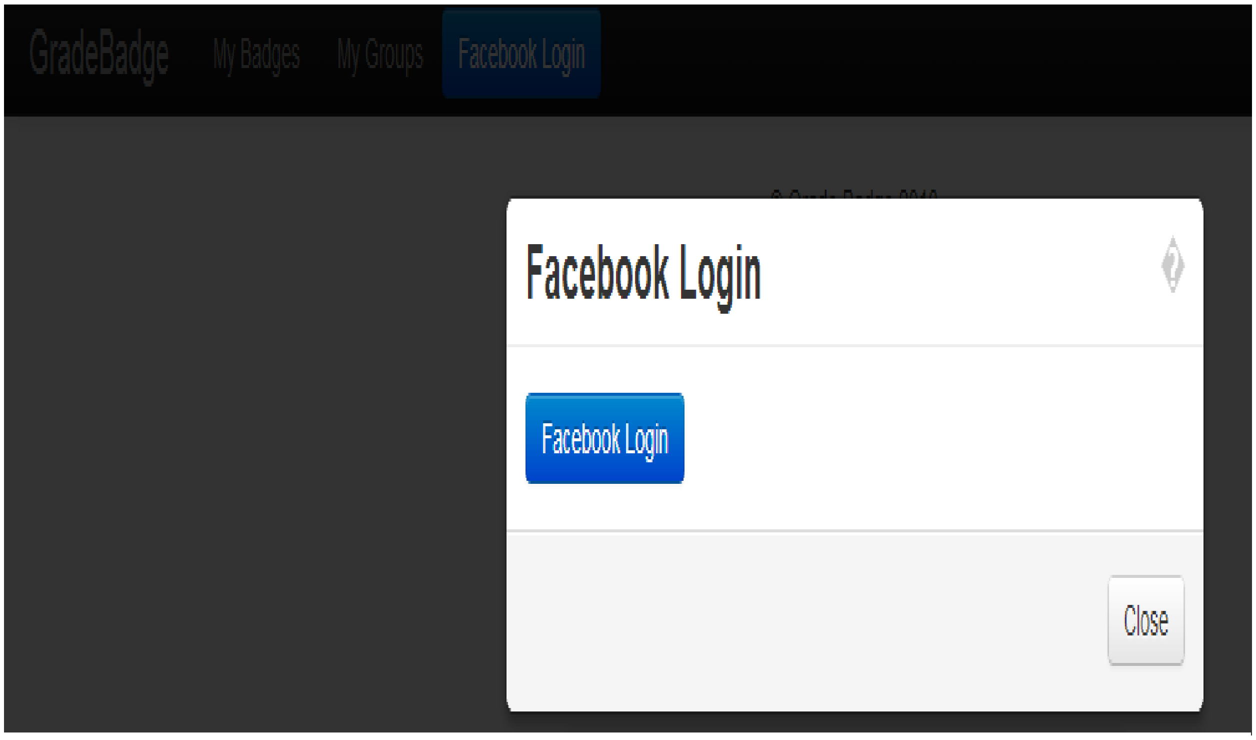
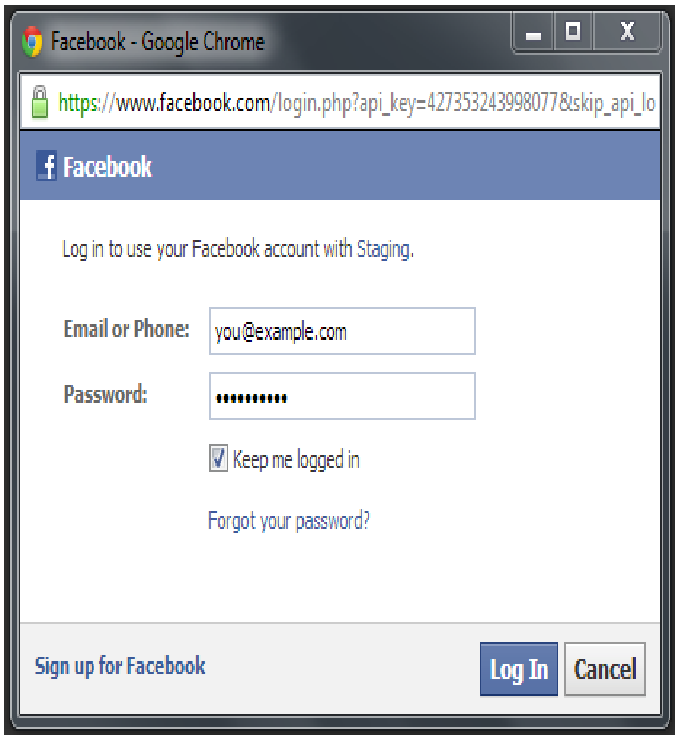


Fig. 6.2: GradeBadge Login Screen



Facebook - Google Chrome
https://www.facebook.com/login.php?api_key=427353243998077&skip_api_lo
Facebook

Log in to use your Facebook account with Staging.

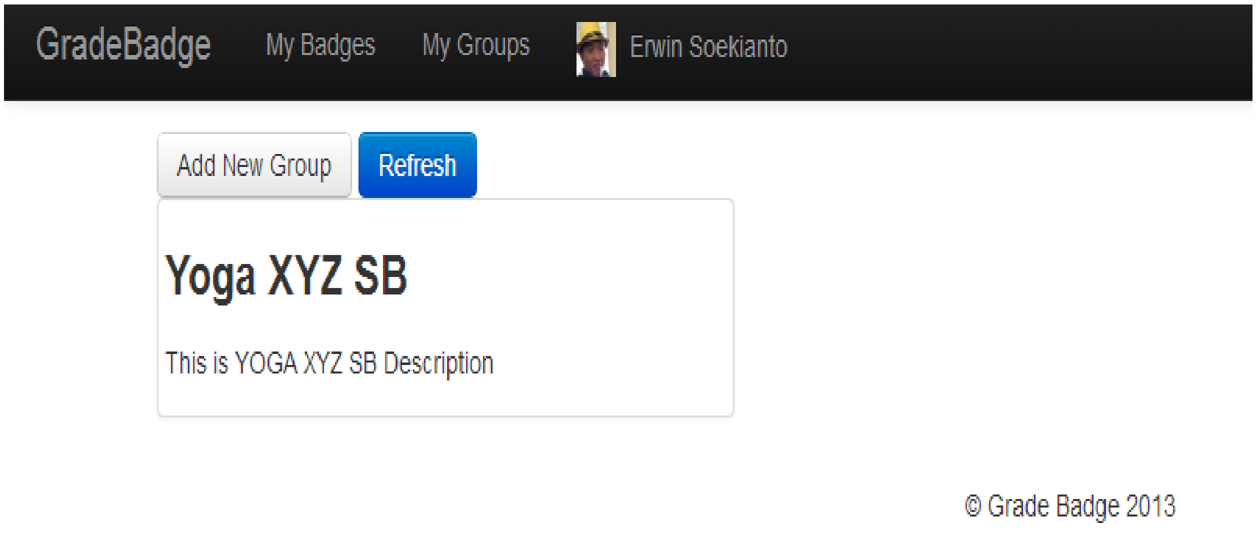
Email or Phone: you@example.com

Password: ••••••••••

[✓] Keep me logged in

Fig. 6.3: Facebook Login Screen

as shown in Figure 6.4.



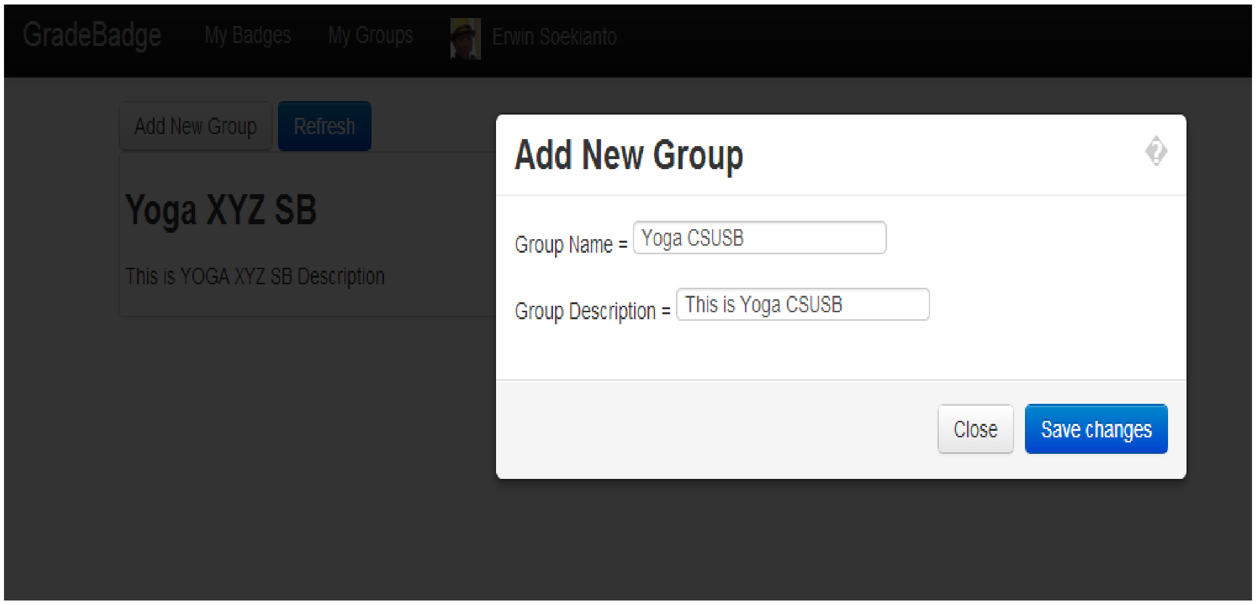
Yoga XYZ SB

This is YOGA XYZ SB Description

© Grade Badge 2013

Fig. 6.4: GradeBadge Group Page

as shown in Figure 6.5.

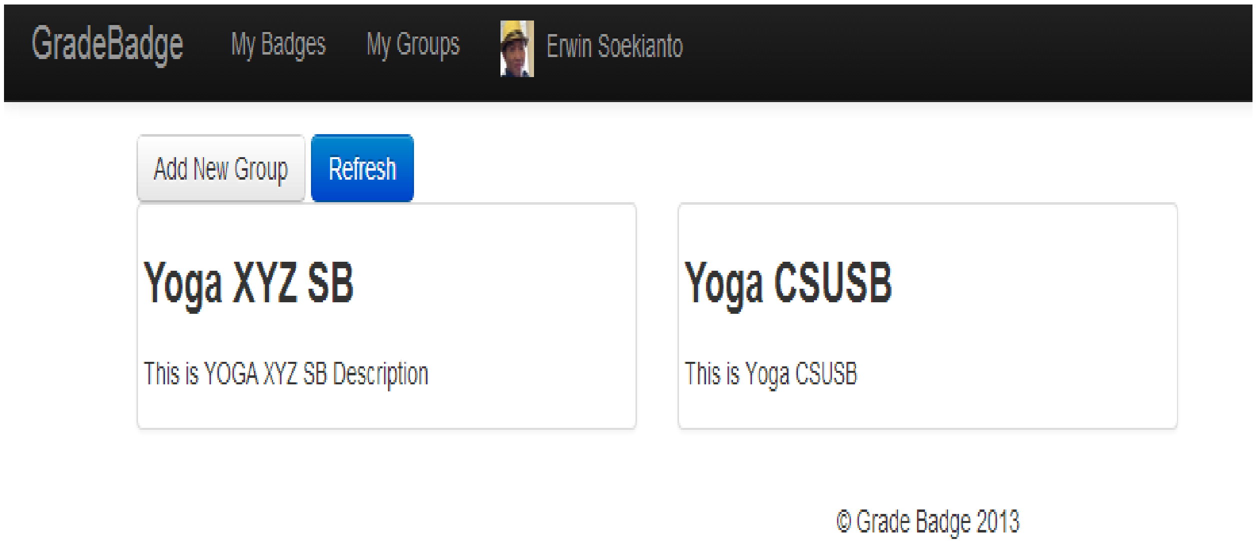


Group Name = Yoga CSUSB

Group Description = This is Yoga CSUSB

Fig. 6.5: GradeBadge Add New Group Page

as shown in Figure 6.6.



Yoga XYZ SB

This is YOGA XYZ SB Description

Yoga CSUSB

This is Yoga CSUSB

© Grade Badge 2013

Fig. 6.6: GradeBadge Group Page Added

7. CONCLUSION AND FUTURE DIRECTION

7.1 Conclusion

GradeBadge is cloud-based web application that is hosted in Heroku, uses Nodejs and MongoDB, that provide responsive design that works well to be run inside browsers in desktop computers, tablets and smart phones. GradeBadge also enables users to login with their Facebook account to simplify authentication, and integrated with their Facebook.

Cloud computing provides significant cost-saving to developers to build application that can be scaled up or down almost instantly to accomodate rapidly changing demand.

7.2 Future Direction

GradeBadge application can be used as sample to showcase the development of cloud-based cross-platform application. This application can also be extended and enhanced in the future, as described in the following

- Auto-sharding: Implementation of auto-sharding in MongoDB to support greater scalability.
- Other social networking: Implementation of authentication and integration with other social networking applications such as Twitter, Tumblr, Google+, etc.
- Native App: Development of native IOS , Android and Windows Mobile Appli-

cation.

- API: Development of API to enable other applications to integrate a reward system module to their applications.

APPENDIX A

SERVER SIDE SOURCE CODE

```
//.env

//.env

FB_APP_ID=466760923387961

FB_SECRET=75e7a042473a989a3b876d3ec8749920

PORT=5000

APP_VER=2

MONGO_URI=mongodb://app:gradebadge123@ds045757.mongolab.com:45757/gb-d

//.gitignore

.DS_Store

node_modules

.env
```

APPENDIX B

MODEL CLASSES SOURCE CODE

```
// model_group.js
var assert = require('assert');
var model = require('./model');

exports.create = function(group, cb) {
  model.db.collection('groups').insert(
    group,
    function(err) {
      model.db.close();
      if (err) return cb(err);
      cb();
    }
  );
};

exports.getByIds = function(group_ids, cb){
  model.db.collection('groups').find({'_id' : {$in: group_ids} }).toArray(
      function(err, groups){
    model.db.close();
    if (err) return cb(err);
    console.log('model_group getByIds groups array = '+ JSON.stringify(groups));
    cb(groups);
  });
};

exports.getAll = function(cb){
  model.db.collection('groups').find().toArray(function(err, groups){
    model.db.close();
    if (err) return cb(err);
    console.log('model_group getByIds groups array = '+ JSON.stringify(groups));
    cb(groups);
  });
};

exports.readGroupMembers = function(group, cb) {
  model.db.collection('group_member_links',{'uid' : true}).find(group).toArray(
      function(err, group_member_links){
    if (err) {model.db.close(); return cb(err);}
    console.log('model_group readGroupMembers group_member_links = '+ JSON.
        stringify(group_member_links));

    var member_ids = group_member_links.map(function(group_member_link) {return
        group_member_link.gid;});
    model.db.collection('users').find({'_id' : {$in: member_ids} }).toArray(
        function(err, members){
      model.db.close();
      if (err) return cb(err);
      console.log('model_group readGroupMembers members array = '+ JSON.
          stringify(members));
      group.members = members;
      cb();
    });

  });
};
```

APPENDIX C

REQUEST HANDLER OPERATION SOURCE CODE

```
//op_read_badges_by_group.js
var model_badge       = require('./model_badge');
var model_group_badge = require('./model_group_badge');
var app_ajax          = require('./app_ajax');

exports.handle = function (data, res) {
  console.log('op_read_badges_by_group  input = ' + JSON.stringify(data));
  var group = { gid: data.gid };
  model_group_badge.getBadgeIdsByGroupId(group, function(bids) {
    if (bids instanceof Error) {
      logger.error(__filename + ' : model_group_badge.getBadgeIdsByGroupId : ' +
           bids.message);
      return app_ajax.error(res);
    }
    console.log('group_badges is read = ' + JSON.stringify(bids));

    model_badge.getByIds(bids, function(badges){
      if (badges instanceof Error) {
        logger.error(__filename + ' : model_badge.getByIds : ' + badges.message)
            ;
        return app_ajax.error(res);
      }
      console.log('badges is read = ' + JSON.stringify(badges));
      return app_ajax.data(res, badges);
    });
  });
};
```

APPENDIX D

CLIENT SIDE SOURCE CODE

```
//public_root\channel.html

<script src="//connect.facebook.net/en_US/all.js"></script>
```

REFERENCES

[1] Ajax (programming). http://en.wikipedia.org/wiki/Ajax_(programming).

[2] Amazon Web Services. http://aws.amazon.com/.

[3] Android Mobile Computing Platform. http://developer.android.com/about/index.html.

[4] Bootstrap. http://twitter.github.com/bootstrap/.

[5] Dom Document Object Model. http://www.w3.org/DOM.

[6] Eclipse Founcation. http://www.eclipse.org/.

[7] Entities, Properties, and Keys. https://developers.google.com/appengine/docs/java/datastore/entities.

[8] Facebook Developers. https://developers.facebook.com.

[9] Github. https://github.com/.

[10] Glossary. http://technet.microsoft.com/en-us/library/bb742416.aspx.

[11] Google Developers Academy. https://developers.google.com/appengine/training/intro/whatisgae/.

[12] Heroku How It Works. http://www.heroku.com/how.

[13] How Entities and Indexes are Stored. https://developers.google.com/appengine/articles/storage_breakdown.

[14] Http secure. http://en.wikipedia.org/wiki/HTTPS.

[15] Index Definition and Structure. https://developers.google.com/appengine/docs/python/datastore/indexes.

[16] IOS Mobile Computing Platform. http://en.wikipedia.org/wiki/IOS.

[17] Jetty (web server). http://en.wikipedia.org/wiki/Jetty_(web_server).

[18] Jquery Mobile 1.2 Reference Document. http://www.jquerymobile.com/demos/1.2.0/docs/about/features.html.

[19] Jquery Official Document. http://www.jquery.com/.

[20] Json. http://en.wikipedia.org/wiki/Json.

[21] Mongodb Agile and Scalable. http://www.mongodb.org/.

[22] Mongolab. http://www.mongodb.org/.

[23] Node.js Manual and Documentation. http://nodejs.org/api/.

[24] Nosql. http://en.wikipedia.org/wiki/NoSQL/.

[25] Official Documentation of Google App Engine Java Datastore. https://developers.google.com/appengine/docs/java/datastore/.

[26] Official Java Technology Document. http://www.java.com/en/download/faq/whatis_java.xml.

[27] PolyModel. https://developers.google.com/appengine/docs/python/ndb/polymodelclass.

[28] Unified Modeling Language. http://www-01.ibm.com/software/rational/uml/.

[29] W3C community. http://www.w3.org/TR/REC-html40/.

[30] What is an Object? http://docs.oracle.com/javase/tutorial/java/concepts/object.html.

[31] Windows Azure. http://www.windowsazure.com/en-us/develop/overview/.

[32] Manoj Kulkarni. *GradeBoard: A Cloud-Based Solution for a Student Grading System.* CSUSB - Masters Project, 2013.

[33] Ph.D Steve Burbeck. Mvc How to use Model-View-Controller. http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.