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

June 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

June 2013

Approved by:

David Turner, Advisor, 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, called GradeBadge, 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 among users, GradeBadge is integrated with the social networking site Facebook.

As the result of this implementation in GradeBadge using cloud computing, we don't need to spend time on system administration to manage the servers. We also don't need to purchase and maintain the hardware. Deployment and re-deployment are done easily from the local command line. For this reason, our implementation approach is suitable with student projects or small startup companies.

Cloud computing provides significant cost savings to developers when building

applications that can be scaled up or down almost instantly to accomodate rapidly changing demand.

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

LIST OF TABLES

LIST OF FIGURES

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 companies 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 is 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 to 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.

Nodejs 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 [5]. 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 [10]. 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 is a partner of Facebook and so it 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 Nodejs 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 [17].

1.2.4 MongoLab

MongoLab is the cloud computing provider for MongoDB database that was used in this project [18]. 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 [6]. 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 available 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 [2]. The application's user interface was contructed using Bootstrap.

1.2.7 Jquery

The GradeBadge application uses Jquery for AJAX transactions and DOM manipulation [15]. Bootstrap also depends on Jquery.

1.2.8 Nodejs

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. Nodejs is a platform built on

Chrome's JavaScript runtime. It's purpose is to allow construction of fast, scalable network applications. Nodejs 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 [19].

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. For this reason, a prototype of a reward application using badges, called GradeBadge, was developed to illustrate and explore this emerging paradigm.

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 among users, GradeBadge is integrated with the social networking site Facebook.

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 [26]. 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 [7].
- 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 [15].
- 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 [4]. 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 [24].
- HTTPS: Hyper-text Transfer Protocol Secure, which is a secure network protocol used to encrypt data transferred between server and client [11].
- MVC: Model-View-Controller is an architectural pattern used in software engineering to isolate business logic from user interface considerations [27].
- UML: The Unified Modeling Language, which is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems [23].
- Microsoft Azure: Cloud computing platform provided by Microsoft [25].
- Google App Engine: Cloud computing platform provided by Google.
- Amazon Web Services: Cloud computing platform provided by Amazon [28].
- Heroku: Cloud application platform provided by Heroku [10].
- Android: Mobile operating system provided by Google [1].
- IOS: Mobile operating system provided by Apple [3].
- NoSQL: Types of databases that use key-value pairs for storing data unlike traditional relational databases. [20].
- JSON: Javascript Object Notation, which is a data representation format using key-value pairs [16].

- Ajax: Asynchronous JavaScript and XML, which is a method for Web applications to communicate between client browsers and servers [13].
- CDN: Content Distribution Network.
- Slug: The binary deployed to Heroku.

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.

In order to access this application, an internet connection is required. On tablets or smart phones, the internet can be accessed through Wi-Fi, 3G, 4G or LTE networks.

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.

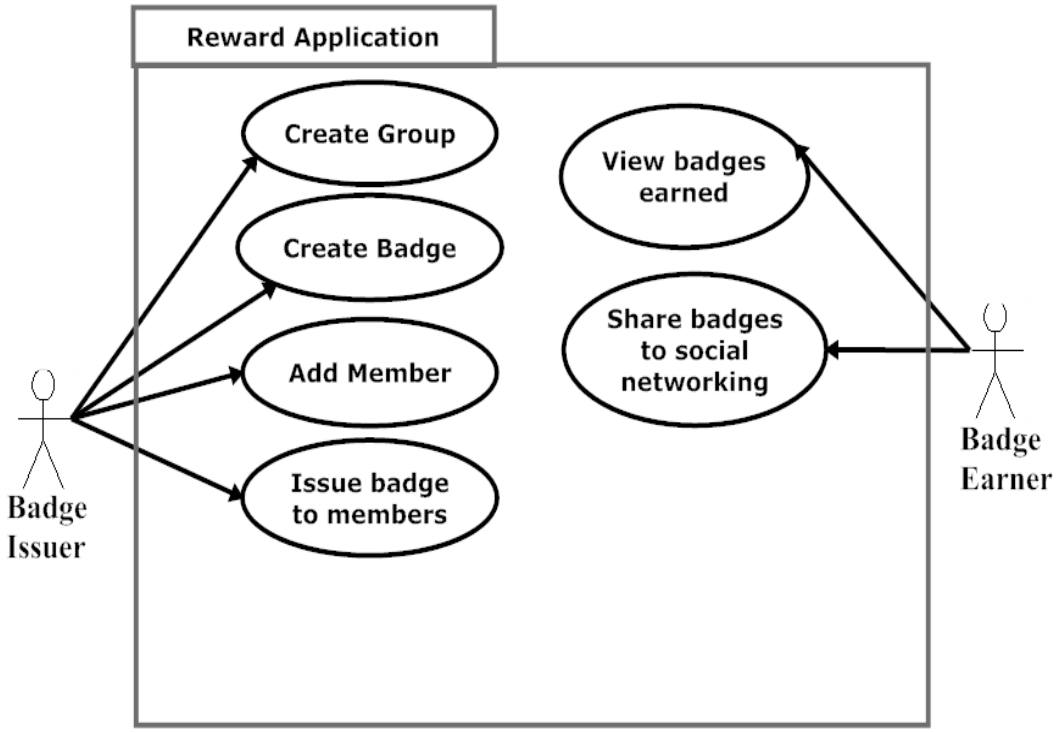

Fig. 2.1: Use Case Diagram

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 internet based, so the down time of the application server is a factor. Because we are using Heroku cloud service, the down time of the server application is a function of Heroku down time. Heroku has reputation of reability, and during the testing of this project, no down time of Heroku was observed.

The speed of page loads is another performance issue. We are expecting page loads to be under 2 seconds consistently.

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 code will be written to adhere the Google Javascript style guide [9].

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.

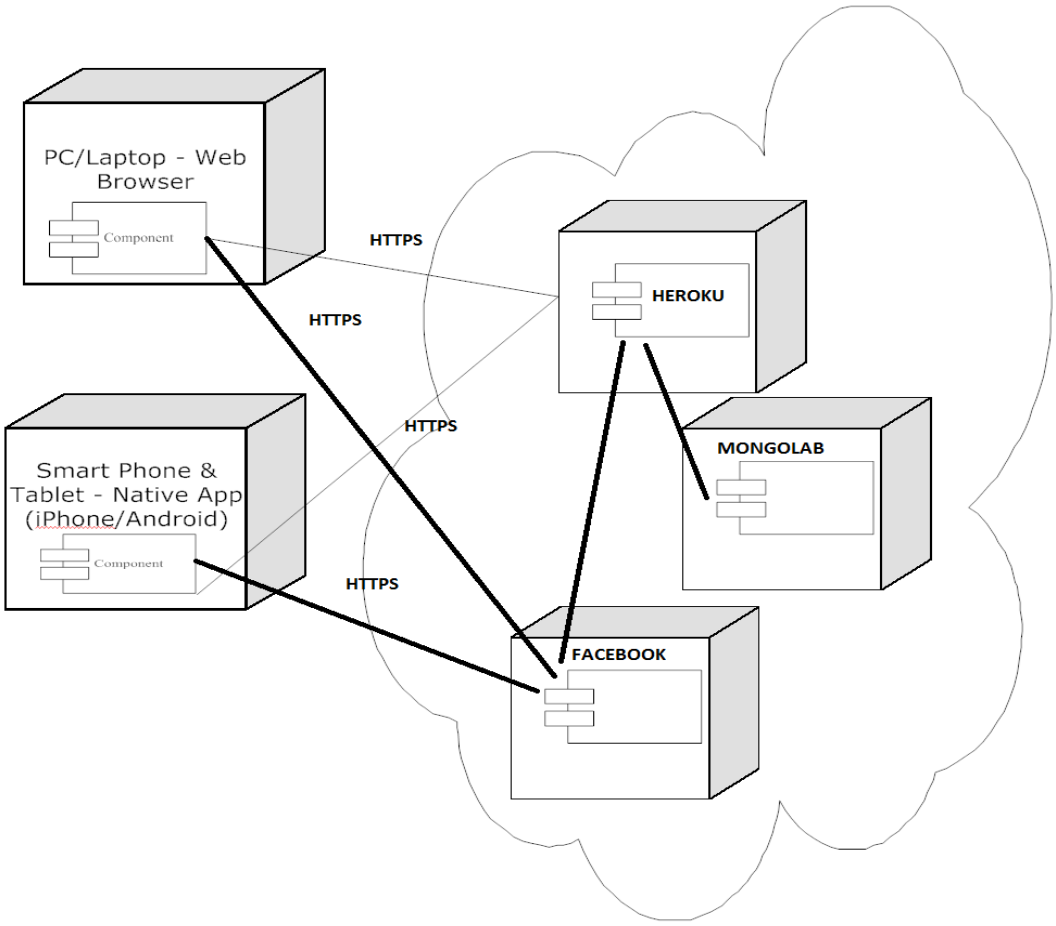

Fig. 3.1: Deployment Diagram

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 deploy-

ment, 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.

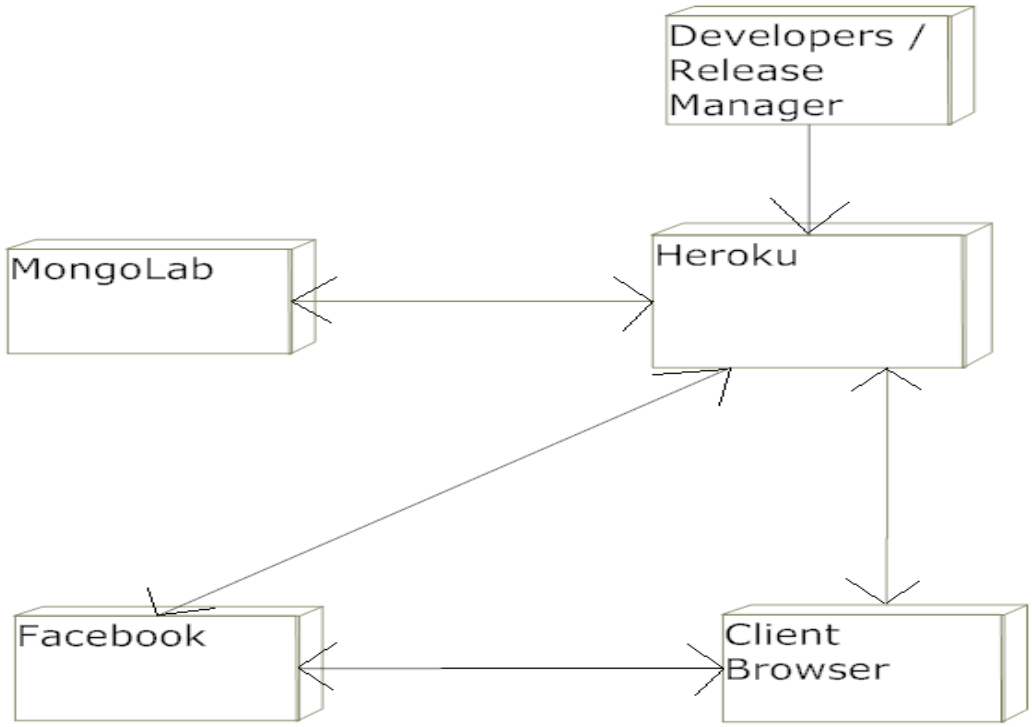

Fig. 3.2: System Integration

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

The GradeBadge application uses a client-server architecture model, and it uses the Model View Controller (MVC) architecture model for both client and server, as shown in Figure 4.1. All communication between client and server is done through AJAX requests submitted through HTTP POST and containing data encoded using JSON.

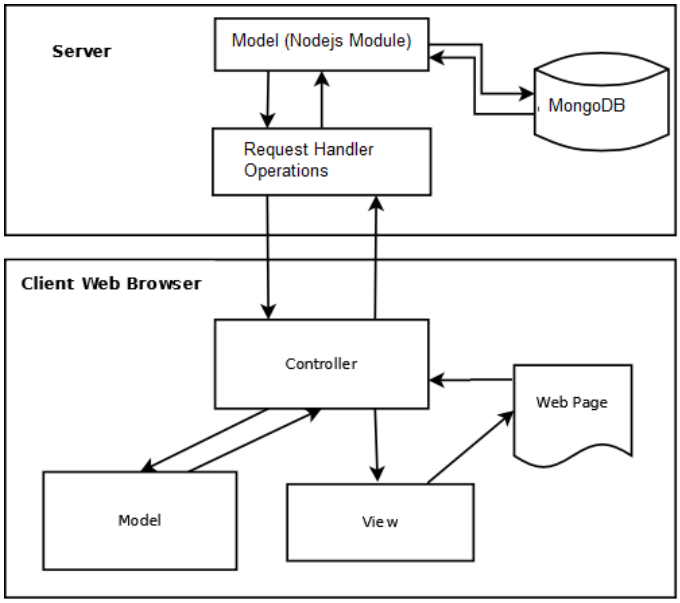

Fig. 4.1: GradeBadge Model View Controller Architecture

4.2 Model View Controller Architecture

This application is based on the Model View Controller (MVC) architecture, which is a software design pattern for separating different components of a software application [27]. There are three main components in the MVC architecture as follows.

- Model: The model includes data in the application and the business rules that apply to it.
- View: The view includes the UI components in the application. The UI components are responsible for presenting the model and for collecting user input.
- Controller: The controller is responsible for updating the data in the model and notifying the view about changes in the model.

4.3 Bandwidth Reduction Strategy

The following sections describe the three strategies used in this project to reduce consumption of bandwidth. Bandwidth is one of the major costs to consider when building a scalable application using cloud computing services, so minimizing bandwidth consumption becomes critical.

4.3.1 File Compression

In this application, in order to reduce the bandwidth consumption, files that are transferred from the application server will be compressed in the gzip format if the client browser supports it, which will result in smaller files. The following code shows how browser requests for static resources are tested to see if the browser supports gzip compression. This code appears in the req_file module.

```
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);
}
```

In addition to checking whether the browser supports gzip compression, the above code also checks for any etag value sent by the browser. The etag functions as a version identifier, which the browser sends to the server with the request message to see whether the file is different from what is in the browser cache and what the server would return. If the etag indicates that the file is current, the server doesn't need to return the file; it simply returns a code indicating that the browser's cached version of the file is current. The following shows how this is done in code.

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

4.3.2 Content Distribution Network Resources

In this application, we rely on content distribution networks (CDNs) to deliver some static resources to browsers without consuming bandwidth to the application server. For example, the application relies on the Bootstrap UI framework. The files for this framework are available through a CDN provided for free to application developers. The following code shows how GradeBadge loads the CSS component of the Bootstrap system.

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

4.3.3 Caching

In this application, we implemented a caching strategy in which every request is checked to see whether a needed file has been modified or is cached in the client browser. This is done by examining an etag value that identifies a version of the requested file. If the browser's version of the file is current, then the server replies with a code of 304, which indicates that the file is current. If the etag does not identify the current version of the file, then the server return the current version of the file. This strategy is implemented to reduce unnecessary consumption of bandwidth.

The following describes the four possible responses from the server when the browser requests a file. The source code is taken from the app_http modul.

- Reply Not Found: This is the response when the requested file is not found; the server returns the code 404, as shown in the following code snippet.

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

- Reply Not Modified: This is the response when the requested file is not modified as determined by comparing the browser's etag with the server's etag. The following shows how this is done in the code. Note that the server takes this opportunity to reset the expiration time of the requested file to one year later.

  ```
  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 response when the requested file is not cached in the client browser. The following shows how this is done in the code, where the server responds by returning the requested file.

```
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 response when the requested file is cached in the client browser, in which case the server will reset the expiration date of the file to one year later. The following shows how this is done in the code.

```
    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
  });
```

4.4 Server-side Design

4.4.1 Request Processing

In Nodejs, components are organized into modules, which can function as namespaces. In this project, all Nodejs modules that start with req_ are request handlers that get requested from the router module. All Ajax requests will go through the req_op module, which verifies that the user is logged into Facebook and the application version is current. Every request passing through req_op must contain a Facebook access token and application version identifier. If the user is not logged into Facebook, then req_op returns the JSON document login:true. If the application version is not current, then req_op returns the JSON document ver:true, as shown Figure 4.2

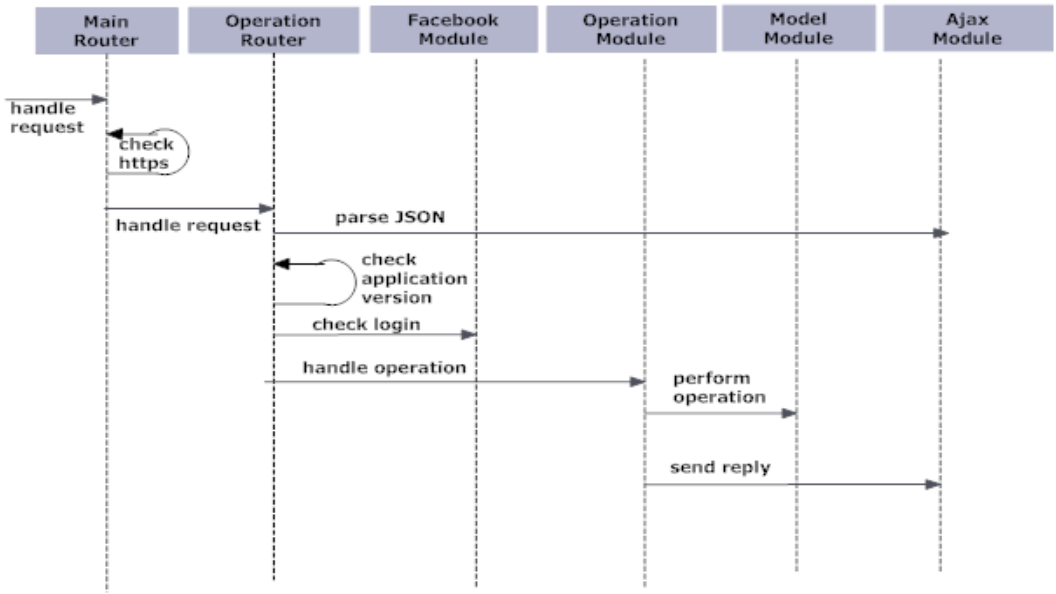

Fig. 4.2: Request Processing Sequence Diagram

4.4.2 Configuration

The following are the list of configuration files required on the server side.

- .env: This is the setup file that contains environmental variables. This file only exists in the developer local environment. The same environmental variables need to be present in the Heroku execution environment, however, they are specified through another mechanism, namely through executing the Heroku config command for both staging and production environments.

  The following is an example of the environmental variable values that could be set in the development environment. In this example, FB_APP_ID is the Facebook application ID and FB_SECRET is the Facebook application secret. (For each environment, we are using a different Facebook application instance.) PORT is the HTTP port number that the server will listen to for connections. APP_VER is the current application version. MONGO_URI is the mongo database connection string. (We use different database instances 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 matching patterns that tell git which files should be ignored (not placed under version control) when committing to a branch of the local repository.
- .slugignore: This is the setup file that contains a 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 a list of Nodejs dependencies and engine version to use when deploying the application. This file also contains the 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
  ```

4.4.3 Nodejs Modules

Below are the list of files that are used when the server executes.

- main.js: This is the main module in Nodejs, which contains code that verifies all necessary environmental variables are set correctly. It also invokes initialization functions of other modules to start the application. After initialization is complete, the main module starts the HTTP request handling loop.

  The following is the part of the code in the main module where it initializes the model, router and fb modules and runs 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 appropriate module. The following is the part of the code in the router module where it checks for pathname of the incoming request and routes the request to a corresponding module request 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. The following is a sample function of the Ajax module, which is used to send data back to the browser in JSON format in the utf8 character encoding.

```
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 the HTTP protocol routines for the application. The use of caching headers, compression headers and other HTTP based optimizations are implemented in this module. This file was discussed in the caching strategy section.

- fb.js: This module contains all code that interacts with Facebook. The following is a sample initialization function, which will check the Facebook App ID and secret.

```
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. The following is a sample of error logging in this module. The application limits the number of error messages printed to avoid server slowdown in the case when there are many errors. A single error is cause for concern and study; recording a continuous stream of errors would be counter-productive and may expose the application more easily to denial of service attacks.

```
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 connection pool during server start up. The following is the code used to establish this connection pool. Note

that it uses the MONGO_URI value from an environmental variable. It also uses the connection options that were set in the model module.

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

- req_app.js: This module handles requests for the application's HTML template for badge earners. The following is the initialization function of the req_app module, where it returns the app.html template, replacing FB_APP_ID with the Facebook application ID prodived through an environmental variable. To conserve bandwidth, the code uses a gzip compressed version of the file and sets the etag value.

```
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 the logging counter. The following is the request handler function of the req_counter module, where it constructs the page with logging information and sends it back to the browser in the utf8 character encoding 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 requests for static content. The request handler function of this module is discussed in the 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 consumption resulting from loading all static resources and compressing them. These resource are kept in memory at all times to eliminate access to the disk drive.

```
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.length;
  }
  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. The following is the initialization function of the req_issuer module, where it returns the issuer.html template, replacing FB_APP_ID with the Facebook application ID prodived through an environmental variable. To conserve bandwidth, the code uses a gzip compressed version of the file and sets the etag value.

```
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 / 1024) + ' MB</p>' +
             '<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.APP_VER + '/");</script>', 'utf8');
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 Nodejs 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.

- model_group.js: this Nodejs module represents Groups collection. The 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} }).toArray(function(err, groups){
    model.db.close();
    if (err) return cb(err);
    cb(groups);
  });
};
```

- model_badge.js: this Nodejs module is used to manage the database collection of badges. The following is one of the functions in the model_badge module to create badge documents in the badge collection from a 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 Nodejs module represents the Users collection. The following is one the functions in the model_user module; it records the login activity of the user by updating the last login timestamp 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 Nodejs module represents the group_admin_links collection, which is used to associate groups and admins through a many-to-many relationship. (A group has one or admins and an admin has one or groups.) The following is one of the functions in the model_group_admin module; it gets the array of group ids from a given admin user through their id.

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

- model_group_member.js: this Nodejs module represents the group_member_links collection. The following is one of the functions in this module; it gets the users that are members of a group.

```
exports.getMemberIdsByGroupId = function(group, cb) {
  model.db.collection('group_admin_links',{'uid' : true})
      .find(group).toArray(function(err, group_admin_links)
      {
    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) {return group_admin_link.uid;});
    cb(member_ids);
  });
};
```

- model_user_badge.js: This Nodejs module represents the user_badge_links collection. The following function of the model_user_badge module is used to create the document that links user with badges (through their ids) to show which badges are own by a given user.

```
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 Nodejs module represents the group_badge_links collection. The following function from this module creates the documents that link groups with badges.

```
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();
    }
  );
};
```

4.6 Request Handler Operation

There will be one Nodejs module to handle Ajax requests from browsers, named op_(request-type).js. Every request may read, write or update to and from more than one collection.

The following are the list of files that define the request handler modules.

- op_read_badges_by_group.js: This operation is for requesting all badges in a given

group. The following shows how this is done in the code, where the operation gets the array of badge ids for a given group id through a group_badge linking document. After getting the badge ids, it gets the badge documents the array of badge ids.

```
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 getting the group documents for a given group admin user. The following shows how this is done in the code, where the operation gets the array of group ids for a given admin id through the group_admin linking document. After obtaining the group ids, the operation gets the group documents from the array of group ids.

```
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 creating a new badge. The following shows how this is done in the code, where the operation adds the badge document in memory to badge collection in the database. After creating a new badge, the operation creates the linking document that associates the badge with a group.

```
exports.handle = function (data, res) {
  var badge = { name: data.name, desc: data.desc, pict:
      data.pict, gid: data.gid };
  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 creating new groups. The following shows how this is done in the code, where operation adds a group document to group collection and creates the linking document that associates the group with an initial admin user.

```
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 Architectural Design

The following is the list of files that are used on the client-side.

- app.html: This is the html template for generating the badge earners Web page. This file contains the HTML skeleton of the HTML page. The following is a list element appearing in the template; this list element will be populated by data after page load by Javascript, which gets the data through an Ajax call to the application server.

```
<ul id="groups_list" class="thumbnails">
  <!-- auto generated -->
</ul>
```

- issuer.html: This is the HTML template for the badge issuer Web page, which is a skeleton from which the page is constructed. The following is a sample of HTML code from this template; it is the list element that contains the logged in user's name. This name is determine after login, and inserted as content by Javascript.

```
<div class="nav-collapse collapse">
  <ul class="nav">
     <li id="name"> <!-- auto generated --></li>
     <li id="login" style="display:none;"></li>
     . . . other menu items . . .
   </ul>
```

```
</div>
```

- public_root/channel.html: This file is static content required by Facebook. The following is the content of this file.

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

- public_root/favicon.ico: This is the favicon image requested by browsers; it is static content.

- public_ver/app.js: This is the client-side Javascript. The following is a sample function, which performs screen transition. When transitioning to a screen, this function calls the screen's init function. The init function may use Ajax to retrieve data from the server needed to present current information to the user.

```
a.screen = function(screenName, speed){
    if (speed === undefined) speed = 300;
    var newScreen = screens[screenName];
    if (newScreen.init) newScreen.init();
    currentScreen.transitionTo(speed, newScreen);
    currentScreen = newScreen;
};
```

5. DATABASE DESIGN

5.1 Overview

5.1.1 MongoDB

MongoDB is a scalable, high-performance, open source, NoSQL document-based database. MongoDB provides a document-oriented storage paradigm, including indexes for fast queries, replication for increased reliability, and auto-sharding for scalability [17].

5.1.2 MongoLab

MongoLab is a cloud computing provider for the MongoDB database that was used in this project [18]. MongoLab has a free tier service, which was used for the purposes of this project.

5.1.3 Documents

Data in MongoDB is stored in documents. Every document must have a primary key named _id and reside in a document collection. Compared to SQL databases, documents in a collection are like rows in a table. Unlike SQL databases, MongoDB documents don't adhere to a schema, other than the requirement they have a _id field that serves as a primary key. When creating a new document, if you don't specify an _id field, mongoDB will add it automatically [17].

Although document structure is not enforced in MongoDB, different data models

and structure may have significant impacts on MongoDB and application performance. So it is good to keep some kind of structure or pattern in the data model [17].

5.1.4 Collections

Documents in MongoDB are organized into collections, and basic database operations are performed relative to collections. Indexes can be assigned to any field or subfield contained in documents within a MongoDB collection; they are defined on a per-collection level. Indices are need to perform queries that run faster than linear time. [17]

5.2 Data Model

In this project, there are three main collections: User, Group and Badge. There are four linking collections to represent the many-to-many relationships between the main collections that are needed by the application. These collections are described in the following, and shown in the Figure 5.1.

- 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 pairs of group ids and user ids, which represent the many-to-many relationships between groups and their admins.
- group_member_links: This linking collection contains pairs of group ids and user ids, which represent the many-to-many relationships between groups and their members.

- badge_user_links: This linking collection contains pairs of badge ids and user ids, which represent the many-to-many relationship between badges and their earners.
- group_badge_links: This linking collection contains pairs of group ids and badge ids, which represent the many-to-many relationships between groups and their badges.

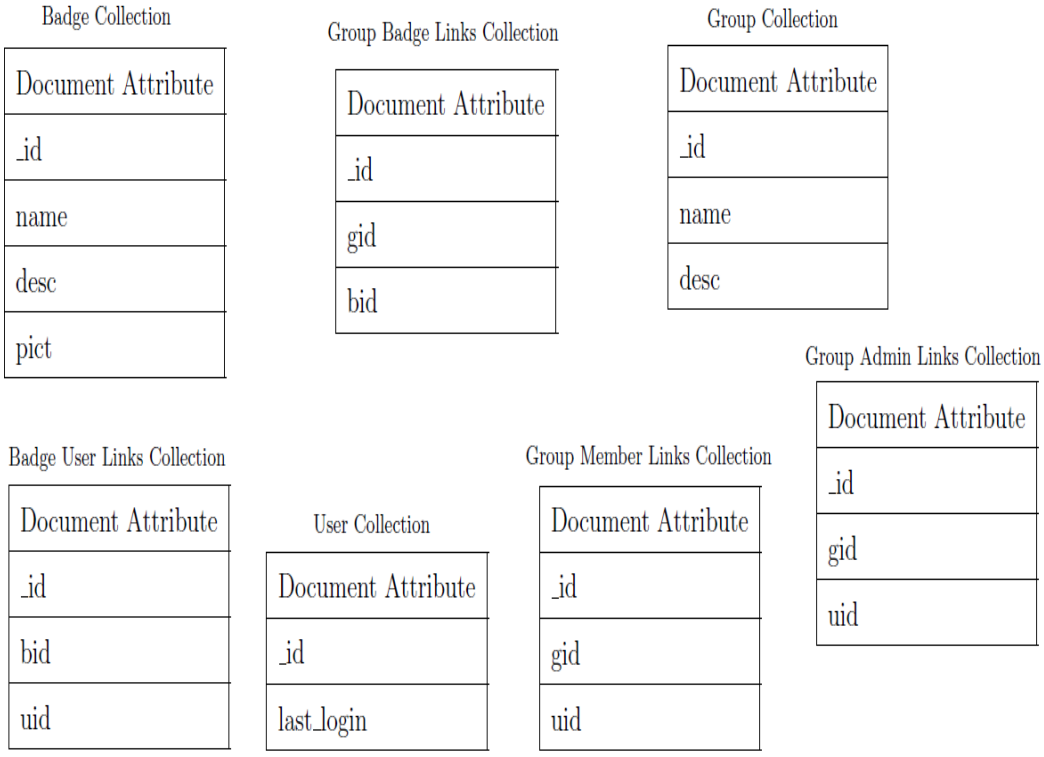

Fig. 5.1: GradeBadge MongoDB Collections

5.2.1 User Collection

The user collection contains user documents, which contain the information shown in 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

The badge collection contains badge documents. Table 5.3 shows the information contained in badge documents.

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

The group-admin link collection contains documents that define the admin users of groups. Table 5.4 shows the information contained in the group-admin linking document.

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

The group-member link collection contains documents that define the member users of groups. Table 5.5 shows the information contained in the group-member linking document.

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

The badge-user link collection contains documents that define the earner users of bagdes. Table 5.6 shows the information contained in the badge-user linking document.

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

The group-badge link collection contains documents that define the badges of groups. Table 5.7 shows the information contained in the group-badge linking document.

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.

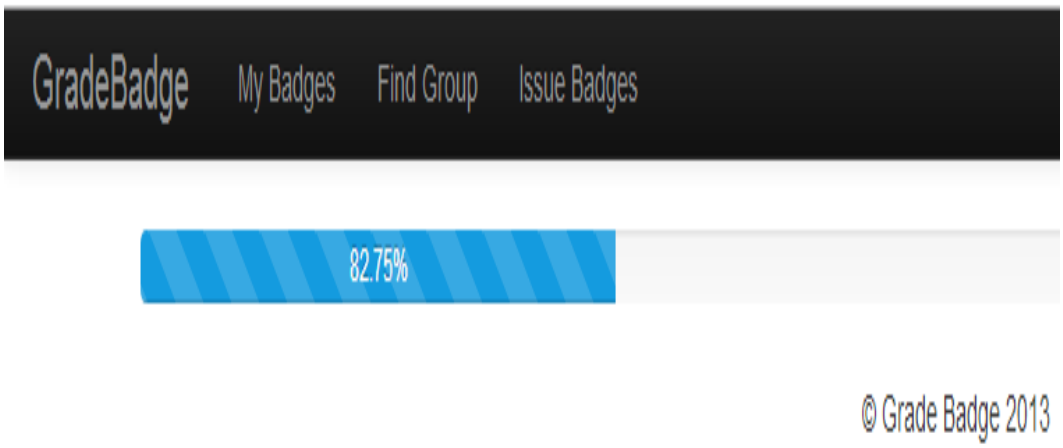

Fig. 6.1: GradeBadge Loading Screen

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.

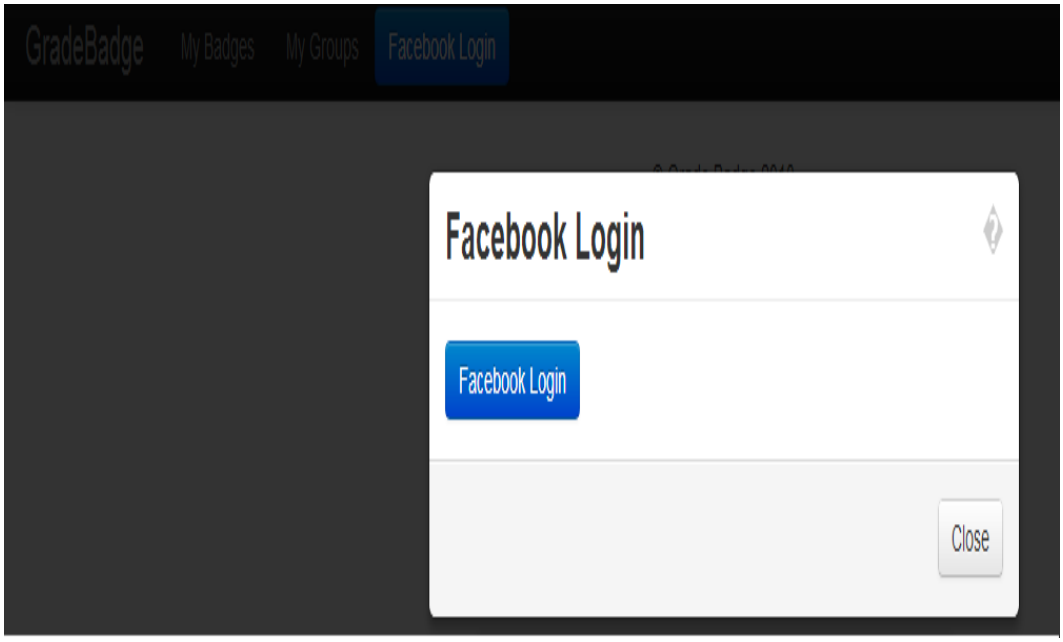

Fig. 6.2: GradeBadge Login Screen

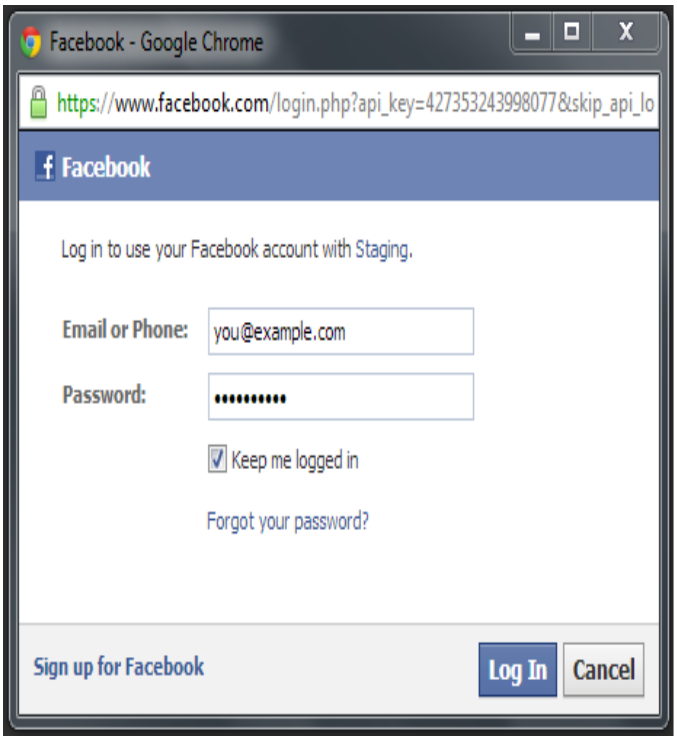

Fig. 6.3: Facebook Login Screen

6.3 Group Page

Badge issuers have access to list of groups that they manage as shown in Figure 6.4.

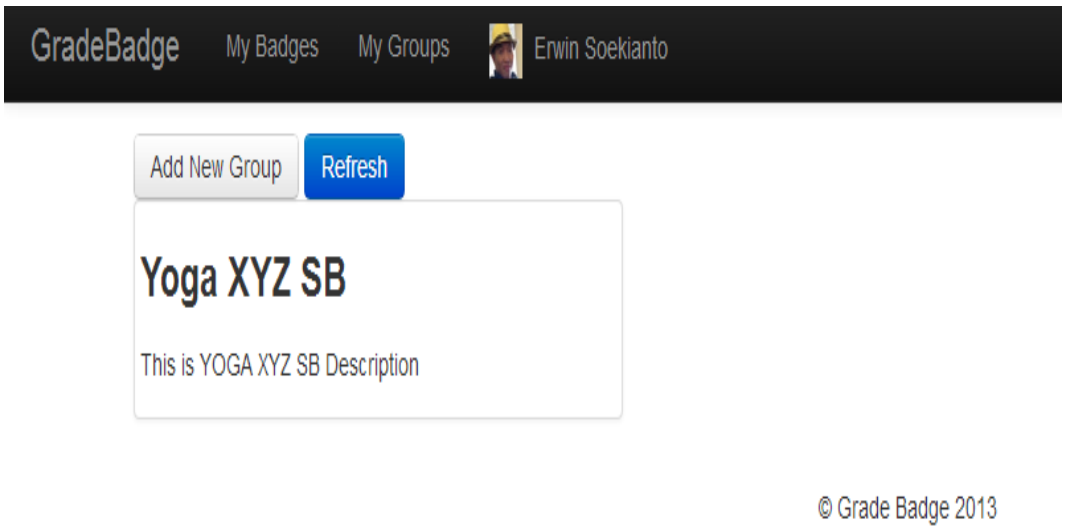

Fig. 6.4: GradeBadge Group Page

6.4 Add New Group Page

Badge issuers can add new group by clicking at "Add New Group" button in group page, then the pop-up modal window to create new group will appear as shown in Figure 6.5.

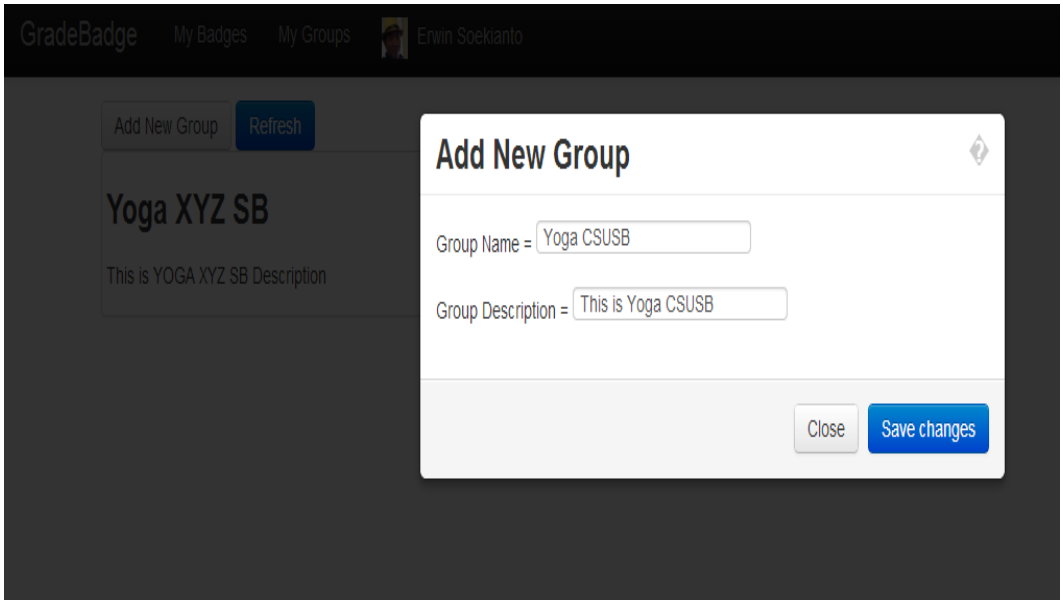

Fig. 6.5: GradeBadge Add New Group Page

New groups have been added to group page as shown in Figure 6.6.

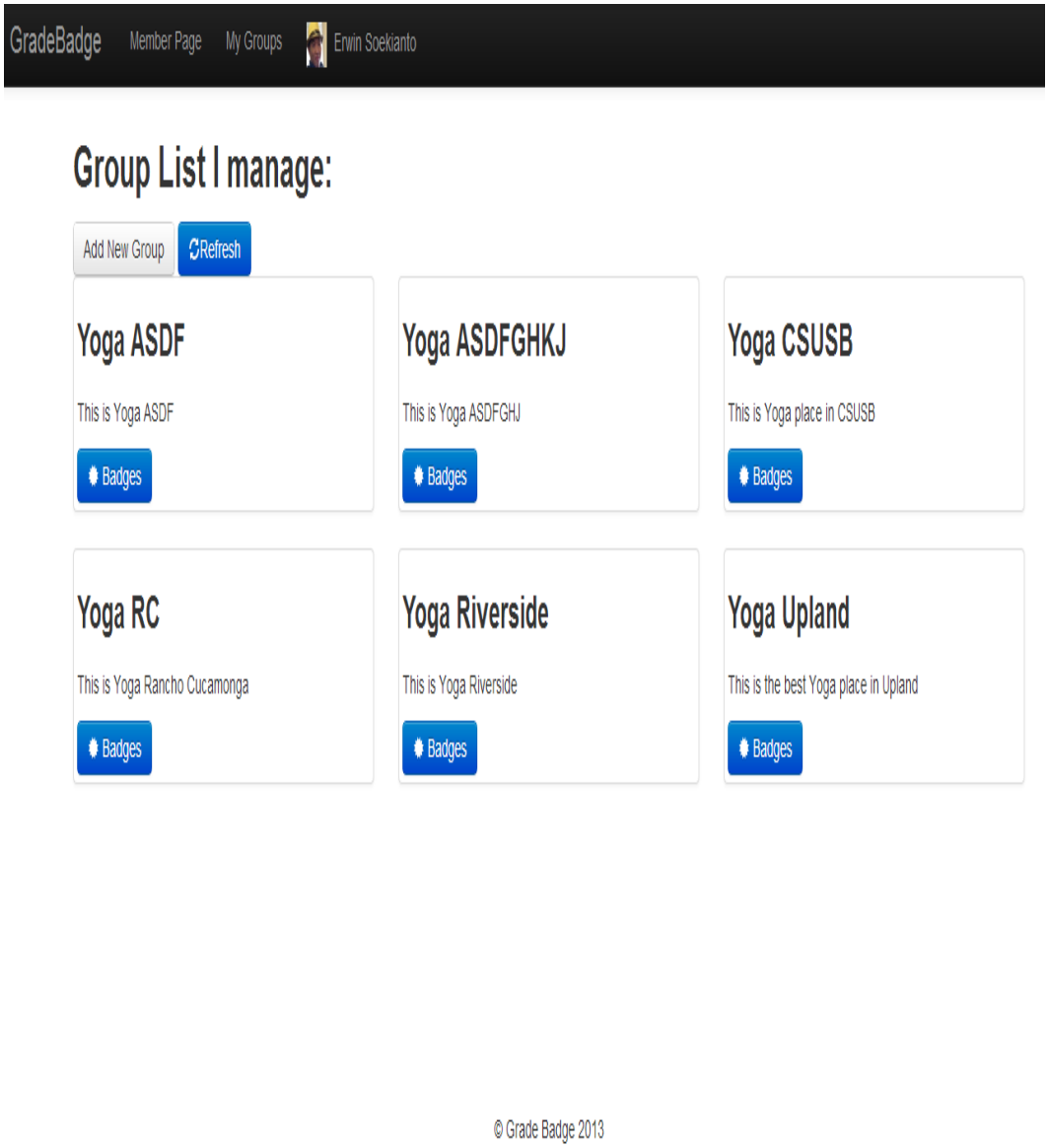

Fig. 6.6: GradeBadge Group Page Added

6.6 Group Badge Page

Badge issuers have access to list badges in a group by clicking at "Badge" button in a group thumbnail, then the application will display the list of badges in the selected group as shown in Figure 6.7.

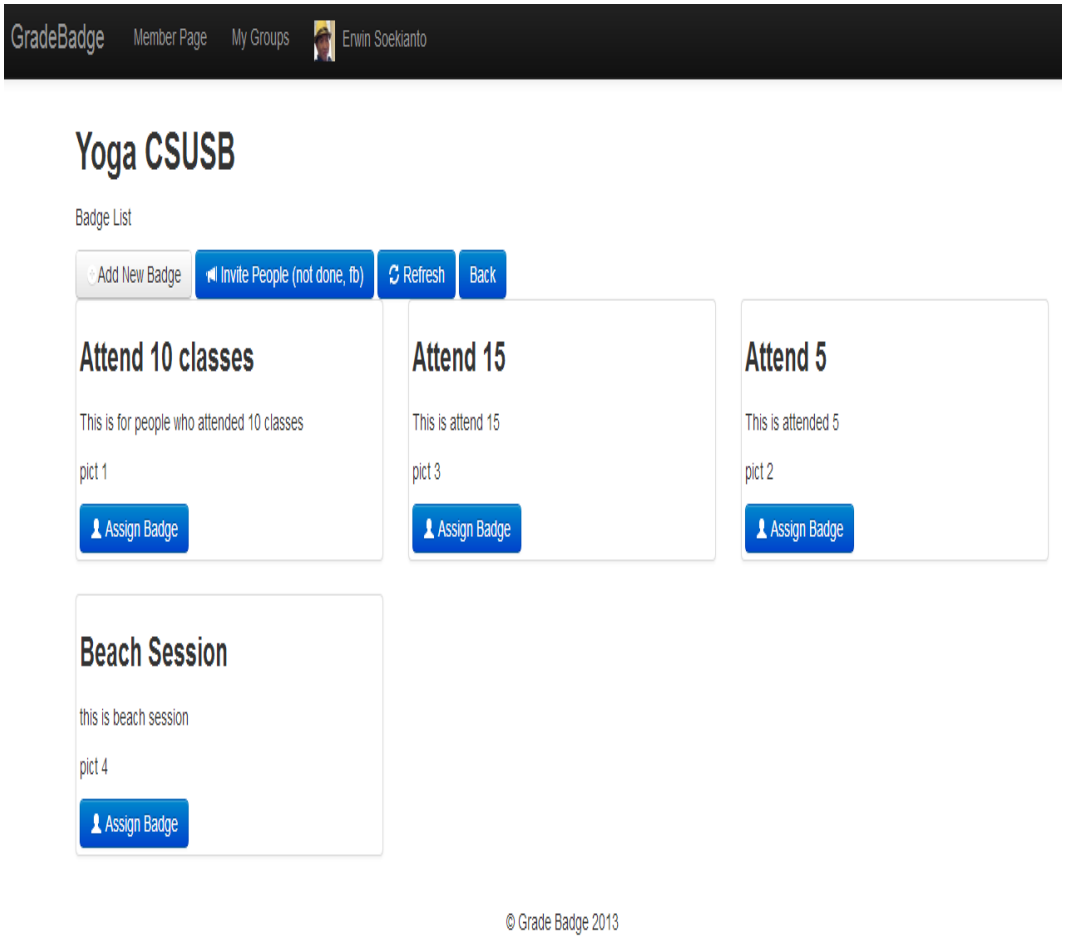

Fig. 6.7: GradeBadge Group Badge Page

6.7 Add New Badge Page

Badge issuers can add new badge by clicking at "Add New Badge" button in badge page, then the pop-up modal window to create new badge will appear as shown in Figure 6.8.

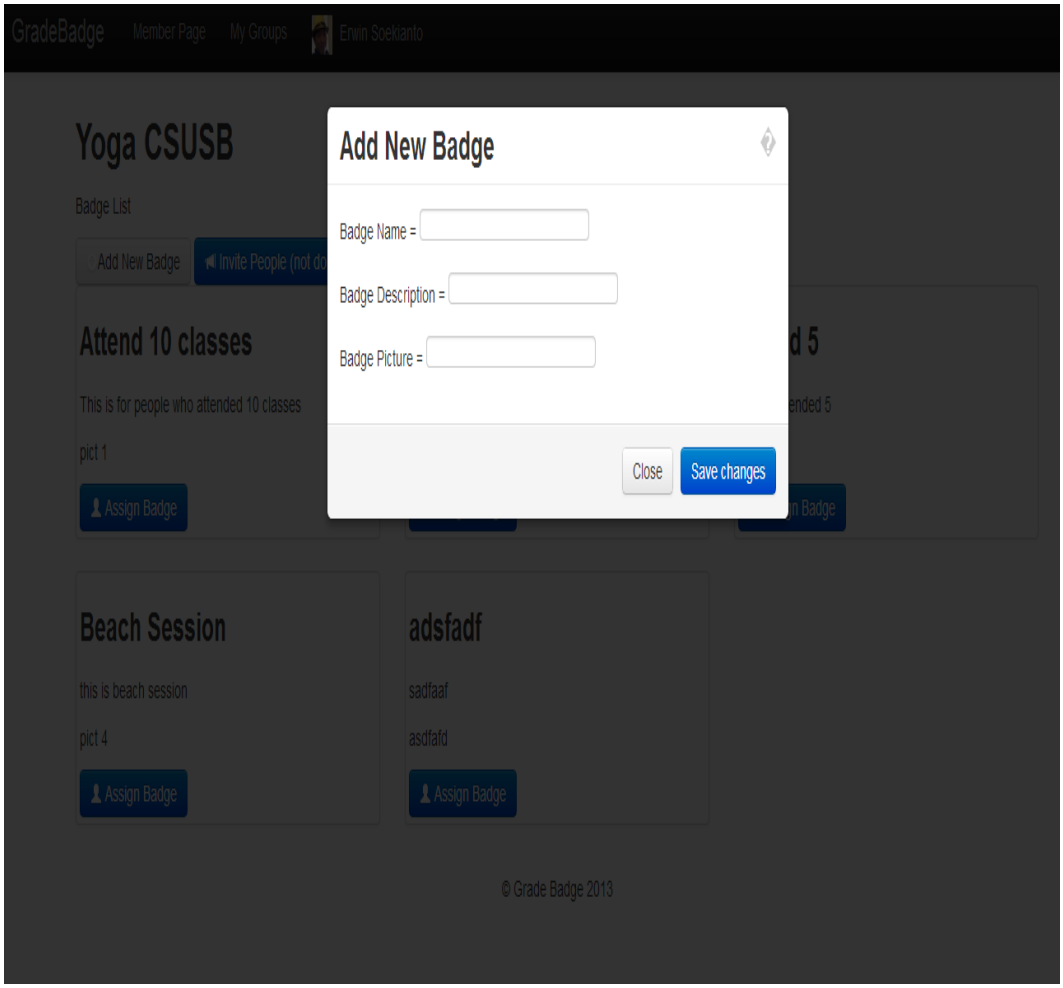

Fig. 6.8: GradeBadge Add New Badge Page

6.8 Logging Counters Page

The application also keeps track of errors, warnings and number of times a function is being callled. It can be accessed through web page as shown in Figure 6.9.

Fig. 6.9: GradeBadge Logging Counters Page

6.9 Memory Statistic Page

The application also monitors the memory and bandwidth usage of the application server. It can be accessed through web page as shown in Figure 6.10.

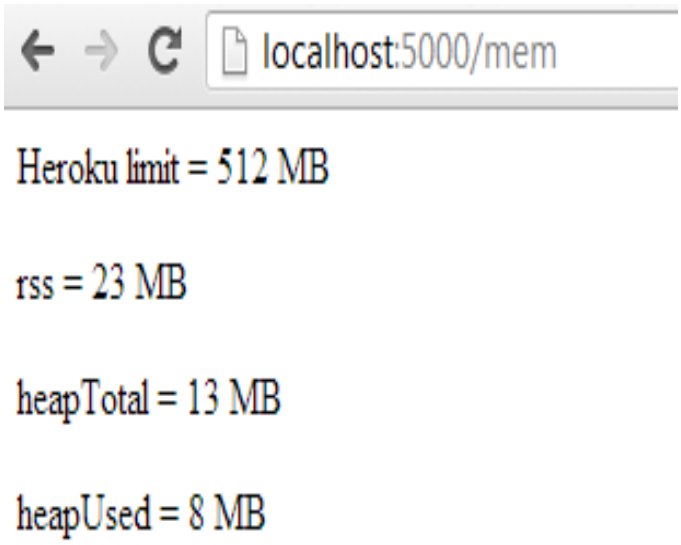

Fig. 6.10: GradeBadge Memory Statistic Page

7. CONCLUSION AND FUTURE DIRECTION

7.1 Conclusion

GradeBadge is a cloud-based Web application that is hosted in Heroku, uses Nodejs and MongoDB, and that uses a responsive Web page design that works well inside browsers in desktop, tablet and smart phone computers. GradeBadge enables users to login with their Facebook account to simplify authentication and provide social networking features such as wall posting of badges earned and friend badge information.

This implementation in GradeBadge uses cloud computing, so developers don't need to spend time on system administration to manage the servers. They also don't need to purchase and maintain hardware. Deployment and re-deployment are done easily from the local command line. For this reason our implementation approach is suitable with student projects or small startup companies.

Cloud computing provides significant cost savings to developers when building applications that can be scaled up or down almost instantly to accomodate rapidly changing demand.

7.2 Future Direction

The GradeBadge application can be used as a sample to showcase the development of cloud-based cross-platform applications. This application can also be extended and enhanced in the future as follows.

- Auto-sharding: Add auto-sharding to the Mongo database in order to support greater scalability. When the data in a collection gets very large, sharding will partition the collection into seperate sections that are stored on different servers. The MongoDB sharding feature automatically distributes and balances data across shard servers.

- Other social networking: Implement authentication and integration with other social networking applications such as Twitter, Tumblr, Google+, etc. This will better serve users who prefer to use other social networking systems.

- Native App: Develop native versions of the application for iOS, Android and Windows Mobile. Even tough GradeBadge application can be accessed via Web browser from tablets or smart phones, it is also important to have a native version of the application, because a native application would provide more responsive user interface.

- API: Develop an API to enable other applications to integrate a reward system module. This will allow GradeBadge to used by larger number of users that are using other established applications such as forums, blogs, and other content management systems.

REFERENCES

[1] Android Mobile Computing Platform. (undated). [Online]. Viewed 2013 February 14. Available: http://developer.android.com/about/index.html.

[2] Bootstrap. (undated). [Online]. Viewed 2013 March 12. Available: http://twitter.github.com/bootstrap/.

[3] Developer for IOS - Apple Developer. (undated). [Online]. Viewed 2013 March 22. Available: https://developer.apple.com/technologies/ios/.

[4] Dom Document Object Model. (undated). [Online]. Viewed 2013 February 6. Available: http://www.w3.org/DOM.

[5] Facebook Developers. (undated). [Online]. Viewed 2013 March 8. Available: https://developers.facebook.com.

[6] Github. (undated). [Online]. Viewed 2013 March 23. Available: https://github.com/.

[7] Glossary. (undated). [Online]. Viewed 2013 March 1. Available: http://technet.microsoft.com/en-us/library/bb742416.aspx.

[8] Google Developers Academy. (undated). [Online]. Viewed 2013 January 31. Available https://developers.google.com/appengine/training/intro/whatisgae/.

[9] Google Javascript Style Guide. (undated). [Online]. Viewed 2013 March 14. Available: http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml.

[10] Heroku How It Works. (undated). [Online]. Viewed 2013 March 6. Available: http://www.heroku.com/how.

[11] Http secure node.js. (undated). [Online]. Viewed 2013 March 7. Available: http://nodejs.org/api/https.html.

[12] Index overview - mongodb. (undated). [Online]. Viewed 2013 March 2. Available: http://docs.mongodb.org/manual/core/indexes/.

[13] Jquery ajax. (undated). [Online]. Viewed 2013 March 2. Available: http://api.jquery.com/jQuery.ajax/.

[14] Jquery Mobile 1.2 Reference Document. (undated). [Online]. Viewed 2013 February 23. Available: http://www.jquerymobile.com/demos/1.2.0/docs/about/features.html.

[15] Jquery Official Document. (undated). [Online]. Viewed 2013 March 2. Available: http://www.jquery.com/.

[16] Json. (undated). [Online]. Viewed 2013 March 3. Available: http://www.json.org/.

[17] Mongodb Agile and Scalable. (undated). [Online]. Viewed 2013 March 13. Available: http://www.mongodb.org/.

[18] Mongolab. (undated). [Online]. Viewed 2013 March 14. Available: http://www.mongodb.org/.

[19] Node.js Manual and Documentation. (undated). [Online]. Viewed 2013 March 8. Available: http://nodejs.org/api/.

[20] Nosql database. (undated). [Online]. Viewed 2013 March 2. Available: http://nosql-database.org/.

[21] Official Documentation of Google App Engine Java Datastore. (undated). [Online]. Viewed 2013 March 15. Available: https://developers.google.com/appengine/docs/java/datastore/.

[22] Official Java Technology Document. (undated). [Online]. Viewed 2013 March 1. Available: http://www.java.com/en/download/faq/whatis_java.xml.

[23] Unified Modeling Language. (undated). [Online]. Viewed 2013 March 10. Available: http://www-01.ibm.com/software/rational/uml/.

[24] W3C community. (undated). [Online]. Viewed 2013 February 21. Available: http://www.w3.org/TR/REC-html40/.

[25] Windows Azure. (undated). [Online]. Viewed 2013 March 11. Available: http://www.windowsazure.com/en-us/develop/overview/.

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

[27] Ph.D Steve Burbeck. Mvc How to use Model-View-Controller. (undated). [Online]. Viewed 2013 March 4. Available: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.

[28] Amazon Web Services. (undated). [Online]. Viewed 2013 March 3. Available: http://aws.amazon.com/.