

San Carlos Regional Headquarter

Computer Engineering Department

Componentes y comunicaciones en internet

Professor: Dennis Valverde

Second programmed project:

Audits Backend

Alonso Vega, 201042592  
Gabriel Fallas, 201030952

Tuesday, May 6th, 2013

Santa Clara, San Carlos

Contents

[Problem Description 2](#_Toc355637146)

[Problem Solution 3](#_Toc355637147)

[Database Design 3](#_Toc355637148)

[Frontend modifications 4](#_Toc355637149)

[Backend functionality 5](#_Toc355637150)

[Non-functional requirements 6](#_Toc355637151)

[Result Analysis 7](#_Toc355637152)

[Conclusions and Recommendations 8](#_Toc355637153)

[Bibliography 9](#_Toc355637154)

# Problem Description

Starting from the ending point of the Audits Software developed in the first part of the project, we now have to implement a backend, introducing the use of Node.js as a server, and Mongo DB as the database engine.

This backend addition must be implement through all the existing project, injecting some functionality in the older version. The idea behind this is to provide a persistent storage for the data being manipulated within the software.

Besides, we’ll include the use of the Node server which is a new form of implementation that has a greater performance in loading times for small projects, this due to the fact that Node requires implementation of every single feature, instead of loading a bundle of functionality as apache does.

# Problem Solution

## Database Design

The database engine is Mongo DB. We implemented the following structure of collections, within the “Audits” database.

* Users
  + Name
  + Password
* Headquarters (HQs)
  + Name
  + Buildings
    - Name
    - Rooms
      * Floor
      * Assets
* Audits
  + Room
  + Asset list
    - Comment
    - State
    - Rating
    - Present
  + Comment

The user collection contains information about each user. Right now, we’re just storing the credential values (e.g. username and password). This information is used to check it against when the website user tries to log in.

The headquarter collection is a great object full of data about each headquarter, with its name and its buildings, each one with its own name and list of rooms. Each room also has information about the floor it is on and its corresponding asset list.

The last collection is about the audits made by the user. This is basically the only collection that gets to be modified at this point of the project. It contains the information previously used dynamically within the page code, modified whenever the user changed any input field.

## Frontend modifications

To match the database using, we had to alter the page code controller. Instead of always saving to the local JSON object, we needed to update the database record.

From the user login page, we implemented the functionality to make an AJAX call each time the button is pressed. Whenever this happens, the name and password input is taken and send within the URL to be handled by the server. The response is a Boolean value, in a string format. The JavaScript code evaluates if the condition to continue is met. If it is, then it navigates to the main page, if it’s not it stays in the same page and shows a dialog box showing the error.

In the main page, the first thing to do is to retrieve headquarter, building and room information from the database. To do this, the page sends an AJAX request to get the array of objects containing the data. The response, if correct, is returned as the string representation. The controller then parses it into a JSON object to be handled locally and used to fill the HTML Selects.

The audit page has to be changed to reflect every change into the database audit collection. We needed to map each value to its corresponding property in the database. This was complicated from the first approach, but became easier later. Every time the save button is clicked or tapped the current audit, local object is send to the database via AJAX to be stored as an update of the record.

From the history view, as well as from the audit page, the user can remove an audit using a gesture or a button. This is sent to the server via AJAX too, and managed by the web server to modify the database and remove the record.

## Backend functionality

The Node.js backend is formed by several modules and sections, each one controlling a specific part or type of request.

The web server is started on a selected port. Whenever a new request arrives, the server checks if it’s a database operation. These functions will come with the “db” path name. When this is the situation, a case expression will check the “do” query string to see what function should be called. The validate method is invoked when the login page calls it. Then there’s the get headquarter method to obtain the list of headquarter containing every building and room in the database. There’s also functions to manage the audits: insert, update and delete.

These functions are stored within a model file. This file includes the database connection with the array of collections it should or is able to edit. Most functions here take a function parameter, named callback to call when an asynchronous function is used. This callback is passed from the web server controller and usually is an anonymous function that writes the response with the results of the database asynchronous call.

Apart from the database, the web server has to deal with the other requests: the website files. These files include the HTML pages, CSS and JS files, images and icons. This is done using a file stream reader that uses the path name to find the respective file and load it into the response.

## Non-functional requirements

The software is developed with mobile support by using JQuery Mobile library, version 1.3.0.

It also includes a CSS file that contains different media queries as part of the responsive design to show an adequate user interface in each device.

The web server is implemented using Node.js. It handles the requests and returns the responses adequately. It also writes using several header contents, such as images (PNG and GIF), CSS and JS files.

The database engine is supported by MongoDB. Using the MongoJS module to connect from Node.js.

The server handle requests for “db” using the model. Once a model function is called, the parameters and a callback is used to return and write the response asynchronously.

# Result Analysis

The final results are presented in the following table; each part has a state column that indicates how complete it is.

|  |  |
| --- | --- |
| Functional and non-functional requirements | State |
| Login page: user validation | Completed |
| Main page: loading headquarters, buildings and rooms from database via AJAX | Completed |
| Main page: creating new audit automatically creates a new record in the audits collection | Completed |
| Audits page: dynamically updating to match the database record with every change | Completed |
| History page: delete from database on gesture | Completed |
| Node.js web server | Completed |
| MongoDB database engine | Completed |

All the functional parts indicated include the view part and do the list of requested conditions.

# Conclusions and Recommendations

From this project we were able to interact with several new technologies, named:

* Node.js: this was the web server that handled every request. We had to learn how to make several things from within it. It was kind of confusing at first, mostly because we are used to the apache web server that automatically manages the requests pathnames and returned the files or processed PHP procedures. Node.js is a more manually or do-it-yourself server. We needed to learn how to return a HTML, CSS, JS or image file stored in the webserver root directory when it was called from the path name. For this we had to learn how to use the file stream reader, just like a C++ or Java program. With this functionality we were able to run the page as a normal page. But then we had to implement the database functions. This was another challenge with Node.js because we didn’t know how to manage the database request and had to understand the driver methods. This included the asynchronous calls with the need of learning the JavaScript asynchronous paradigm. Finally we managed to fit every little knowledge within the two main controllers (webserver and model).
* MongoDB: the NoSQL database engine was another big challenge. As we were used to thinking in tables and relationships between keys and foreign keys, we were constantly facing database design problems. Finally we manage to think every object as a potentially one big bulb of data, as the headquarter information ended being stored; or as a simple record, like the ones used in the users or audits collections.

# Bibliography

* Node.js API Docs. <http://nodejs.org/api/>
* MongoDB 2.4 Manual. <http://docs.mongodb.org/manual/>
* MongoJS API Documentation. <https://github.com/gett/mongojs>