# 1. Introduction

This report outlines the design of our application ‘Craic’ and covers the database design, the server-side logic, the client-side logic, user functionality and the interaction between the client and server. Using Rodriguez (2008) as the theoretical foundation, the report also addresses the RESTful architecture of our application. The client-side is coded as a single-page web application which provides functionality using Asynchronous JavaScript and XML (AJAX). Despite the misleading name, JSON was used as the transport mechanism for data. The server and database were programmed using node.js/express and mongoDB respectively.

# 2. Client-side design

## 2.1 Page structure

The web page consists of a navigation bar at the top of the page and four content boxes in the main body of the page. The content boxes are as follows:

**Box 1:** Information about the currently logged in user, i.e. a constantly displayed profile box. This box is positioned on the left-hand side of the page.

**Box 2:** Information about the user that is being currently viewed, i.e. a dynamically displayed profile information box. This box is positioned in the upper middle of the page.

**Box 3:** Messages that the user who is currently being viewed has on his/her timeline. This box is positioned in the lower middle of the page, and is to be seen in connection with the box above it, which is why they both have been placed in the middle of the page.

**Box 4:** A complete list of users on the right-hand side of the page.

## 2.2 Communicating with the server-side/back-end

Craic is a single-page application, meaning that all of the content is populated dynamically using event handlers and AJAX requests to communicate with the server. AJAX requests are made using jQuery’s ajax() method, specifying the request’s type (i.e. GET or POST), url (e.g. /login), data to be sent to the server (for POST requests only) and success function which is called when the request succeeds. Self-explanatory variable names have been used where possible to give a clear understanding of what content is being sent to the server. The response returned by the server is always the same array regarding its structure, so as to facilitate the process of populating the web page with the relevant content. The structure of the array is as shown in Appendix **A.1**. The returned array’s content is then processed and, using jQuery, used to dynamically populate the web page with the relevant information.

## 2.3 Application logic and functionality

Inspired by Twitter, Craic allows users to do the following: Users can register and login, post messages, upload profile pictures of themselves, look for other users of the application, mark messages as favourites, reply to messages (resulting in a threaded conversation), and spread/share messages. Also, users can follow other users (making ‘friends’), which will result in them seeing all of their friends’ messages on their own timeline. They do not need permission to follow other users. Unfollowing friends will reverse the effect, removing a user’s former friends’ messages from his/her timeline. Messages are displayed in chronological order, with the most recent one being on top. Also, you cannot follow friends you are already following, and you cannot unfollow friends you are not currently following. Similarly, you cannot follow yourself. This is achieved by hiding the respective buttons corresponding to the current friendship status.

# 3 Back-end / Server

## 3.1 MongoDB database design

### 3.1.1 Entity-relationship model

As shown in Figure 1 below, the database is structured such that each user holds multiple messages, but each message can only be posted by one user.

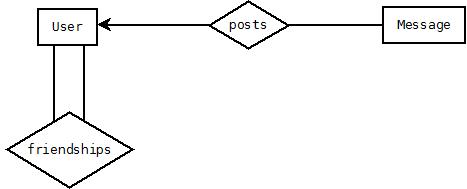


Figure 1 - Craic MongoDB design

Also of note is that Craic users are related to each other by friendships and allow ‘following’ and ‘followers’ to be modelled. For example in the situation where User A follows User B, User A will see all messages posted by User B, but not vice-versa.

### 3.1.2 Table Schema

A database was created using MongoDB that contains three collections as listed in Appendix **A.2**. The reason for creating more than one collection is to minimise any redundant data being stored. MongoDB has the benefit of storing nested data – such as arrays or objects – which is exploited by the database design of the Craic application. In some situations however, it was necessary to separate data by semantic meaning and in a sense perform a join much like what is seen in a Relational Database Management System (RDBMS). Modelling friendships would be very difficult to achieve using only the ‘users’ collection, hence the inclusion of the ‘friendships’ collection.

## 3.2 Node.js server design

We wrote the server-side script to follow the principles of RESTful API from the beginning; requests would be made from the client side through POST or GET requests to different URLs, and these would call the functions that perform the required operations. In order to keep this simple we generally assigned one function to each HTTP request, with the exception of /register which uses two.

When more complex functionality was required, or several operations had to be performed in sequence, we utilised callback functions. The best example of this is the series of functions that are called after retrieveData(request, response), which each add more information to the response object, before returning it. This was used so that the client side script had access to all of the necessary information to build the front-end (cf **A.1**).

A connection to MongoDB is opened each time we needed to access something new. When we had gotten our initial application working, we noticed that many connections were left open, sometimes over 100 if we were using the application for a few minutes. Upon further investigation we found that MongoDB was opening a pool of five connections, and only closing one, each time, which seemed very wasteful. We decided to only open one connection each time manually, and after some testing found that all connections were closed after use.

We used a session variable to store information on which user is logged in, so that the page persists after the user refreshes the page, or navigates away and back to the page again. This session variable is deleted in the logout functionality.

In order to promote security, we hash passwords that users provide on registration, using the password-hash package. Another function from this package is used to verify the submitted password with the stored one when the logging in.

# 4. RESTful API

According to Rodriguez (2008), a RESTful API fulfils the following criteria:

1. Use HTTP methods explicitly.
2. Be stateless.
3. Expose directory structure-like URIs.
4. Transfer XML, JavaScript Object Notation (JSON), or both.

**1)** By making use of the jQuery.ajax() method, our application uses HTTP methods explicitly by using the HTTP GET method to retrieve resources, and the HTTP POST method to create new resources, fulfilling 1). According to The jQuery Foundation (n.d.), PUT and DELETE are not supported by all browsers when used by the jQuery.ajax() method, hence we did not use them so as to ensure the application’s functionality. See Appendix **A.3** for a complete overview of how the methods are being used in our application.

**2)** Our application is stateless in terms of the client side sending “… complete requests that can be serviced independently of other requests” (Rodriguez, A., 2008), and the server can, using the requests, reconstruct the client state. However, our application does rely on a session/cookie combination to enhance usability. This is something where we see room for improvement for future RESTful API implementations.

**3)** The API is designedby using intuitive and directory structure-like URIs within which the AJAX requests are made using jQuery.ajax(). For instance, to register, the specified URL is /register, and to log in, it is /login. See Appendix **A.3** for a complete list of the URLs used by our application.

**4)** Our application fulfils 4) by using MongoDB, which uses JSON documents to store records (MongoDB, Inc., n.d.), and by transferring these JSON documents between server and client.

# 5. Instructions

The entire application is enclosed in the ‘application’ folder in the submission. In order to function correctly, the craicServer.js file must be stored beside node.exe and a ‘content’ folder, which contains the client side script, HTML page, CSS file, and dependent images, as well as two subfolders; ‘images’ that contains user profile pictures and ‘background’ which contains the different options for background (See Appendix **A.4**).

We have also included a pre-loaded ‘mongodata’ folder inside of mongodata.zip with some users that can be used to begin in an environment with some pre-existing users, or alternatively a blank folder can be used to start from a blank scenario. For the blank scenario, the ‘images’ folder should be emptied as well. The order of starting up the database and server should be: **1.** Connecting to the database (e.g. mongod --dbpath H:\CS5003\mongodata), **2.** Server (node craicServer.js) and, optional, **3.** open the mongo shell to check the connection and database (mongo). node.js and MongoDB have to be installed prior to running the application.

As mentioned in the Dependencies section, our application uses the ‘express’, ‘mongodb’, and ‘password-hash’ middleware that must be installed before running the script (e.g. npm install password-hash).

# 6. Dependencies

We use the password-hash package to salt passwords before storing them in the database, so this needs to be installed by running $ npm install password-hash, as well as express and mongodb that were installed from the beginning.

# References

MongoDB, Inc. (n.d.) *What is a JSON database?* [Online] Available from: http://www.mongodb.com/json-and-bson [Accessed 3rd April 2014].

Rodriguez, A. (2008) *RESTful Web services: The basics*. [Online] Available from: https://www.ibm.com/developerworks/webservices/library/ws-restful/ [Accessed 3rd April 2014].

The jQuery Foundation (n.d.) *jQuery.ajax()*. [Online] Available from: https://api.jquery.com/jQuery.ajax/ [Accessed 3rd April 2014].

# Appendix

|  |  |  |
| --- | --- | --- |
| Index | Content | Description |
| 0 | Object  description: string  firstname: string  lastname: string  loggedInUserID: integer  picUploaded: boolean  userID: integer  username: string | Information about viewed user  User’s description  User’s first name  User’s last name  Logged in user’s user ID  User’s picture upload status  User’s user ID  User’s username |
| 1 | integer | Viewed user’s number of messages/craics |
| 2 | integer | Viewed user’s number of followers |
| 3 | integer | Viewed user’s number of friends |
| 4 | Array  Object  \_id: integer  firstname: string  lastname: string  picUploaded: boolean  username: string | Array containing all registered users  One object per user  Unique user ID  User’s first name  User’s last name  User’s picture upload status  User’s username |
| 5 | Array  Object  \_id: integer  date: string  favouritedBy: array  replies: array  text: string  userID: integer  username: string | Array containing all messages to be displayed  One object per message  Unique message ID  Message date  Array of user IDs  Array of objects (one per reply), each object containing two strings, namely *replyAuthorUsername* and *text*  Message text  User ID of message author  Username of message author |
| 6 | Boolean | Friendship between logged in user and viewed user |

## A.1 Response array

## A.2 MongoDB database collections

### A.2.1 users collection

|  |  |  |
| --- | --- | --- |
| Attribute | Type | Comments |
| \_id | integer | Unique ID assigned over default objectID created by mongoDB (PK) |
| username | string | Username for login |
| password | string | Password encrypted using password-hash package (npm install password-hash) |
| firstname | string | Holds firstname of user |
| lastname | string | Holds lastname of user |
| description | string | Stores description for the user profile |
| picUploaded | boolean | True if user has uploaded a jpg image; false if they have not (set to false as default) |

### A.2.2 messages collection

|  |  |  |
| --- | --- | --- |
| Attribute | Type | Comments |
| \_id | integer | Unique ID assigned over default objectID created by mongoDB (PK) |
| date | string | Stores JavaScript timestamp date string |
| favouritedBy | array | Holds array of integers which correspond to users.\_id of favouring user |
| replies | array | Holds array of replyObjects (JSON), see below |
| text | string | Holds text for created message |
| userID | integer | Holds reference to the users.\_id field of the user that created the message |
| username | string | Stores the username of the user that created the message |

**replyObject instance**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Type** | **Comments** |
| text | string | Holds data regarding reply |
| replyAuthor | string | Holds username for author that has replied |

Note: each replyObject is held in the messages collection under the replies attribute

### A.2.3 friendships collection

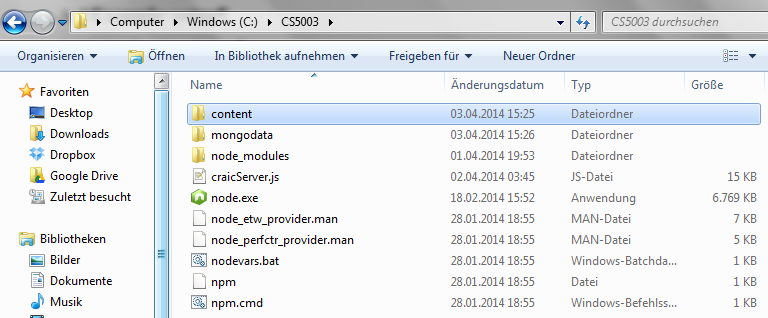
|  |  |  |
| --- | --- | --- |
| Attribute | Type | Comments |
| userID | integer | Id of following user |
| friendID | integer | Id of followed user |
| objectID | string | Default ObjectID created (PK) |

## A.3 RESTful API

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **URL** | **Functions called** | **Description** |
| POST | /register | checkUsername, userRegister | Checks to ensure the desired username is not already in use, and (if not) adds the user’s information to the database |
| POST | /login | userLogin | Verifies the input information against the database, and authenticates the user |
| GET | /sessionCheck | checkSession | Checks for a session variable, and retrieves the data for the logged in user if one exists |
| GET | /messages | retrieveData | Gets all relevant data for the logged in user, or the user’s whose page is opened |
| POST | /postCraic | postMessage | Adds a new message to the database |
| POST | /addReply/:messageID | addReply | Adds a reply to an existing message in the database |
| POST | /favouriteMessage/:messageID | favouriteMessage | Adds the logged in user to the array of who the message is favourited by |
| GET | /userProfile/:id | getUserProfile | Takes the user ID of one of the users and retrieves their data for the response object |
| POST | /followUser/:username | addFriendship | Creates a new entry in the friendships database, whereby the logged in user follows the user in question |
| POST | /unfollowUser/:username | deleteFriendship | Removes the friendship from the logged in user to the user in question |
| GET | /logout | userLogout | Deletes the session variable used for login |
| POST | /uploadFile | saveFile | Takes a JPG from the user and adds it to the /content/images folder with a filename unique to their account, to be used as their profile photo |

# A.4 Example setup

## A.4.1 Application folder



## A.4.2 ‘content’ folder

