Final Report: An implementation of an interactive annotation game on Facebook

Author: Louise Knight
Supervisor: Irena Spasic
Moderator: Paul Rosin
CM0343 (40 credits)

Abstract

The concept of an 'ESP Game' was created by Luis von Ahn in order to solve the problem of finding good tags for images (1). Tags collected through such a game can be used with image search engines and screen readers. However, games like the ESP Game hosted on the GWAP website (2) can suffer from a lack of promotion in that they do not take advantage of social networks, for example, by allowing the user to 'Like' the game on Facebook. We have created, as a proof of concept, a Facebook game that allows for the tagging of images in a way similar to how von Ahn's original game did. Having researched into the technologies available for Facebook app development, we decided to use *Node.js* (3) with *Socket.IO* (4), hosting our app on Heroku (5). Our app allows multiple users to play the game, and also gives the options of posting to one's Timeline or sending Facebook messages to friends about the game, as well as sending requests to friends to invite them to play the game. These functions would encourage more people to play the game, and are further enforced through a system where users earn points for tagging images. Over time, as more and more users play the game, a wider variety of tags would be able to be collected for the images stored on the system.

Acknowledgements

Completing this project would not have been possible without the help and support of my supervisor, Irena Spasic. She has given me much helpful advice throughout the course of this project, as well as very useful feedback on my written work. It is under her supervision that I have been able to develop into an independent researcher, and I feel much more confident in my ability to tackle academic problems - it is my experience of completing this project that has motivated me to pursue a PhD.

I also wish to thank my parents, John and Jean. Their support throughout this project enabled me to carry on when I searching for bugs in my code and juggling this project alongside my other commitments. They also reminded me that sometimes I need to draw a line under my work - relentless perfectionism can be as much of a time-waster as it can be a motivator.

Finally, I give thanks to my best friend and boyfriend, Tom for his encouragement and humour - he has kept me entertained throughout not only this project, but also throughout my time at university.

Table of Contents

Introduction

Since the Interim Report, we have implemented an image annotation game as a Facebook app resembling Luis von Ahn's ESP Game on the GWAP website (2). This game allows multiple users to partner up and collectively solves (at least partially) the problem of finding useful tags for images. Because our game was implemented as a Facebook app, there is the potential for it to reach a wider audience than a game hosted on some other, non-social networking website; the use of functionalities such as posting about the game to one's Timeline, or sending requests to friends, encourages more users to play the game, building up a larger and more varied collection of tags for the images stored on the system.

This report details how the design of our app has changed since its inception in the Interim Report, gives code snippets and explains how the core functionalities of the game were implemented, evaluates the project and its results, then details future work and draws conclusions on the project as a whole. Finally we reflect upon our learning during the project.

A video demonstration of the finished app can be found at: http://youtu.be/B_Xcsnbm0tI (we recommend that this video is viewed in full screen mode). The app itself can be found at: https://apps.facebook.com/tagginggamelouise/

The words ‘app’ and ‘game’ are used interchangeably throughout this report.

Design

Here we shall take each component of our original design (from the Interim Report) in turn and specify how it has changed (or whether it has changed at all).

Choice of architecture

Originally our choice of architecture was a repository style, with the paths to the images stored in a database, and the images themselves stored in a regular file system. The image data is accessed by 'clients' (browsers). This design has changed slightly upon implementation in that, although we are still using a repository architecture such that the images are being stored centrally and accessed by clients, we are not using a file system to store and access the images. The images are instead being stored in 'the cloud' using Cloudinary (6), an online image management tool. This is because Heroku does not allow for the storing of images directly (i.e. in a file system), and so a separate image hosting site was needed. Also, a Cloudinary account was easily set up due to it being offered as an 'add-on' for a Heroku app.

Design decomposition

As in the Interim Report, we shall break down the game into its main functions and describe how they have changed since the Interim Report.

Function "Authorise application"

This functionality has not changed in that the user still needs to read and accept the 'small print' when they access the game for the first time, but instead of being asked to do so before they are allowed to visit the app itself as most apps do (Figure 1), the user needs to click a 'Login' button within the app and then authorise as normal (see Figure 2). The information that is requested of the user includes their public profile, friend list, photos and likes. In the course of a game, we are only using their Facebook ID to identify them as a user across multiple game plays, as well as their friend list in order to send requests and messages promoting the game. In the future, photos could be used when implementing the leader board.

Function "Play a game"

With regards to the algorithm specified in the Interim Report for this function, the approach taken is much the same, except for the fact that if one user in a pair leaves the game, the game will be aborted, with the other user receiving a message telling them that their partner has left the game. This is because one of the pieces of core functionality, the ability to play the game when there is either one user in the system, or an odd number, was not implemented, for reasons discussed later in this report.

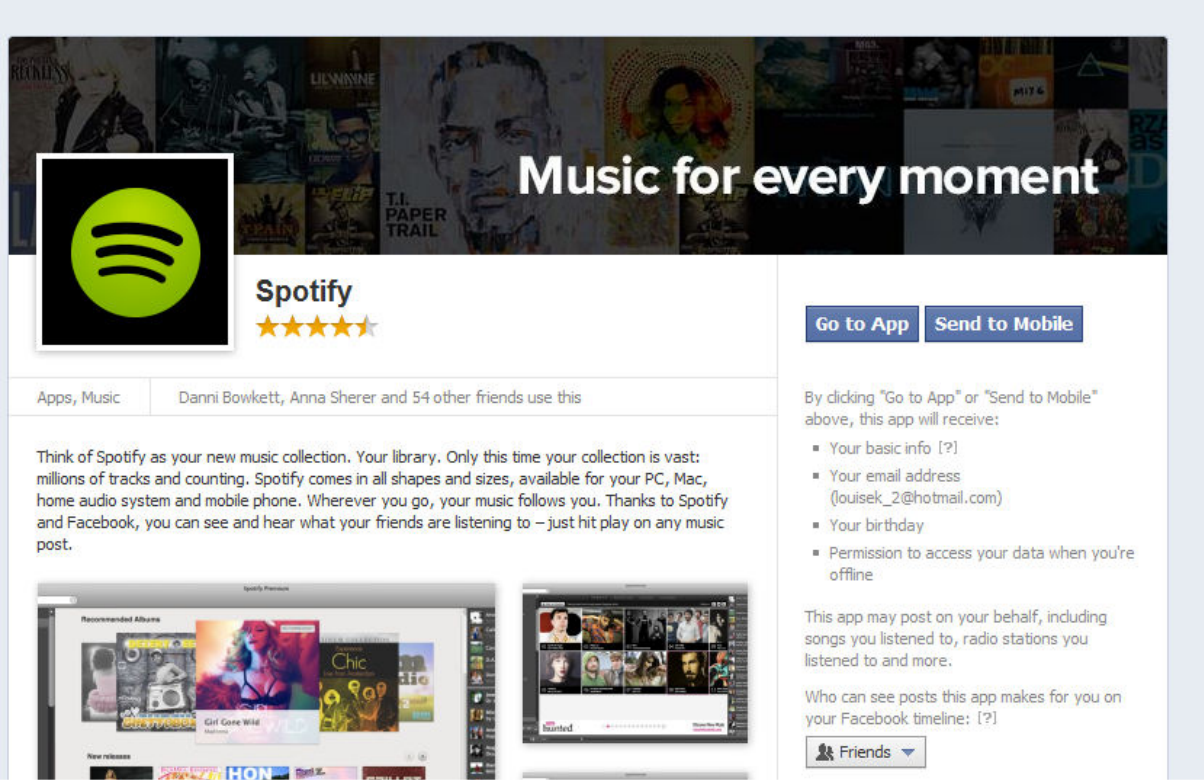


Figure 1: The user is asked to agree to the 'small print' before they are allowed to access the app

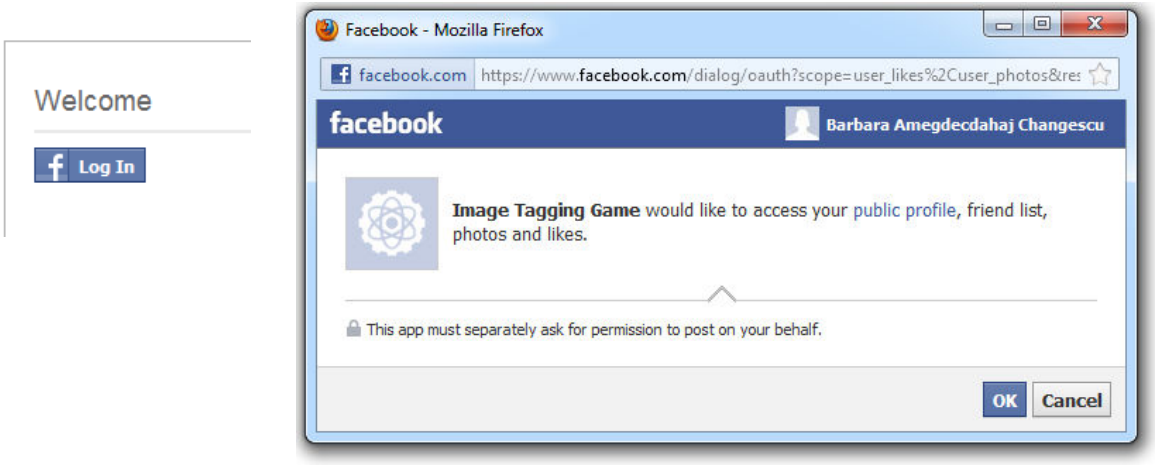


Figure 2: The user has to click the 'Log In' button before they are requested to agree to the 'small print'

Function "Recommend to a friend"

This piece of functionality has been implemented, but not exactly as described in the Interim Report – instead of posting to a friend's Timeline about the game to recommend it, we can send a personal message to that user recommending the game. This allows us to write a personal message to the friend(s) the message is sent to, and it also includes a link to the game (see Figure 3). In addition, included in the Interim Report was an image of being able to send requests using the Request Dialog, which we can also do (see Figure 4). It is these

pieces of functionality that would encourage more users to play the game over time, which would allow us to collect more tags.

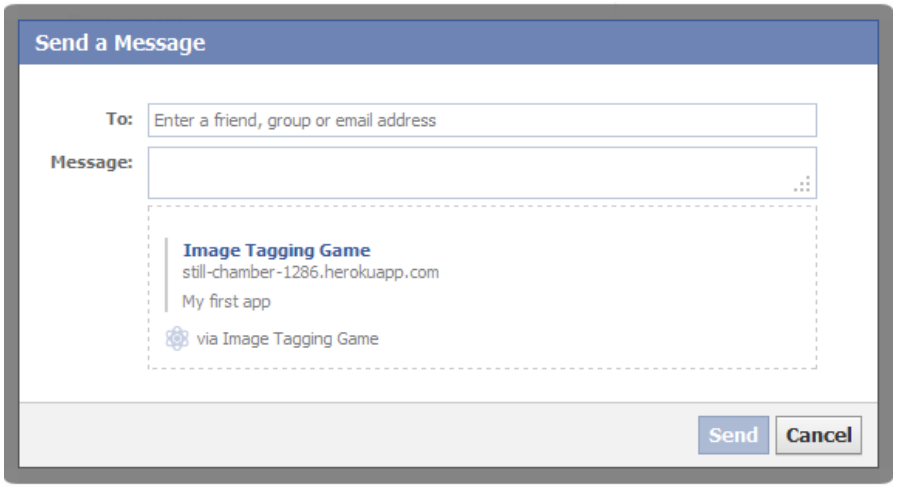
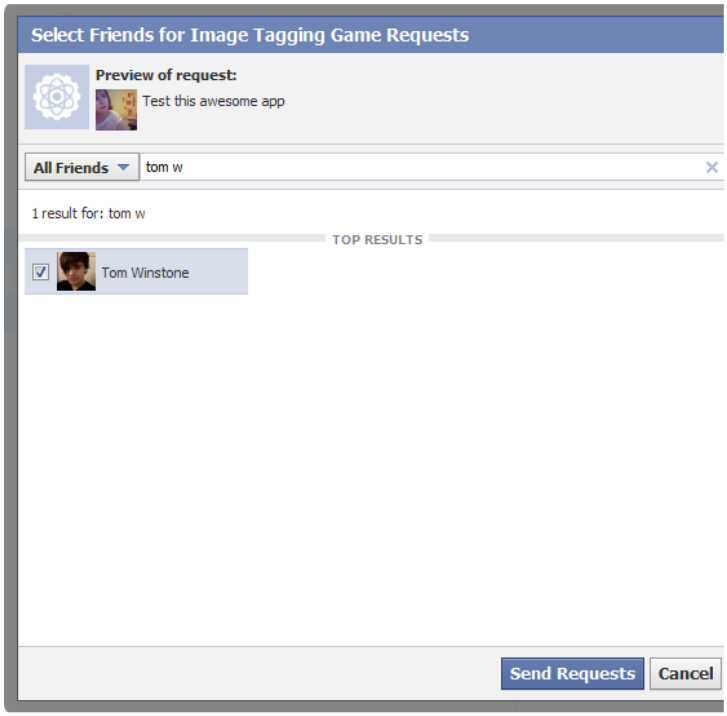


Figure 3: Sending a message to friends with a link to the app



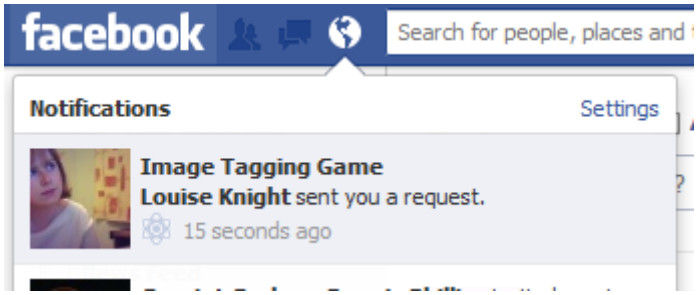


Figure 4: Sending requests to friends to play the game

Functions "View leader board", "Post about high score", "Review tags"

These were optional game functions, and due to problems discussed later in this report, were not able to be completed, although some of the ground work towards implementing these features has already been done. It should be noted that, with regards to posting about one's high score, a user can still post to their own Timeline, but such a message would not include the high score that user has just obtained (but the user can specify whatever message they want) (see Figure 5).

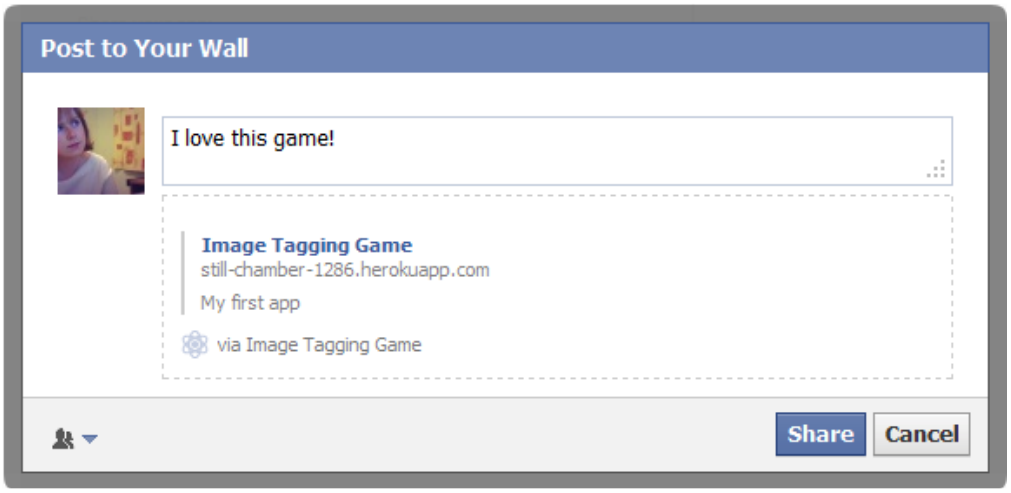


Figure 5: Posting a message to your Wall (or Timeline) about the game, including a link to it

Choice of languages

The technologies specified for the implementation in the Interim Report (*Node.js*/*Socket.IO* with Heroku app hosting and a Heroku Postgres database (7)) were used with no change, with some additional technologies used (for example, Cloudinary for online image hosting).

Database design

The design used in the implementation ended up very similar to the design specified in the Interim Report, with two exceptions. The first is that we are not storing the path to each image since we are not storing the images in a file system (we are instead using Cloudinary, as mentioned previously). The second is that we are not storing the set of tags the last user that saw this image entered, because we did not implement the functionality for a user to play if there are an odd number of users on the system or there is only one user. It was this functionality that necessitated storing the tags entered, as well as the number of seconds into viewing that image that those tags were entered, so that if a user didn't have a partner, the computer could essentially 'become' the partner, playing back previously entered tags for each image. That being said, we did set up the table to facilitate the storing of this data (*previous_guesses*).

Below is the updated database schema:

images (image_id, creator)
non_taboo (image_id, tag_id, agrees_left)
previous_guesses (image_id, seconds, tag_id)
taboo (image_id, tag_id)
tags (tag_id, tag_text)
users (user_id, score, total_score)

The *images* table is used to store the IDs of all of the images, along with the creator of each image; storing the creator is necessary because the website that the images were taken from (FreeDigitalPhotos.net) requires an acknowledgement when the images are published on a website (8). The *non_taboo* table is used to store tags that are not taboo yet; we store the image the tag is associated with, the tag's ID, and the number of 'agrees' left until that tag becomes taboo. *previous_guesses* would be used in the future to store a set of guesses for a particular image, with the number of seconds into viewing that image that they were entered, so that an odd number of people could play the game simultaneously. *taboo* stores the taboo tags with their associated images, *tags* stores a tag ID along with each tag so that we do not need to store many copies of the same tag at once, and *users* stores the maximum score and accumulative score for each user (who is identified by their Facebook ID).

Interface

The original interface design is shown in Figure 6, and the implemented interface is shown in Figure 7.

Although this design is much simpler, this isn't of too much importance due to the fact that this was primarily a project of implementing the idea of the ESP Game as a proof of concept, rather than making the game aesthetically pleasing.

In addition, there is some debugging information displayed at the bottom of the app as the game is played (see Figure 8) – this has been kept to allow us to check that tags are being entered correctly into the database, but of course, it would be removed before the app is released to the public.

In this section we have taken each component of the initial design of our system and discussed how it has changed through the implementation.

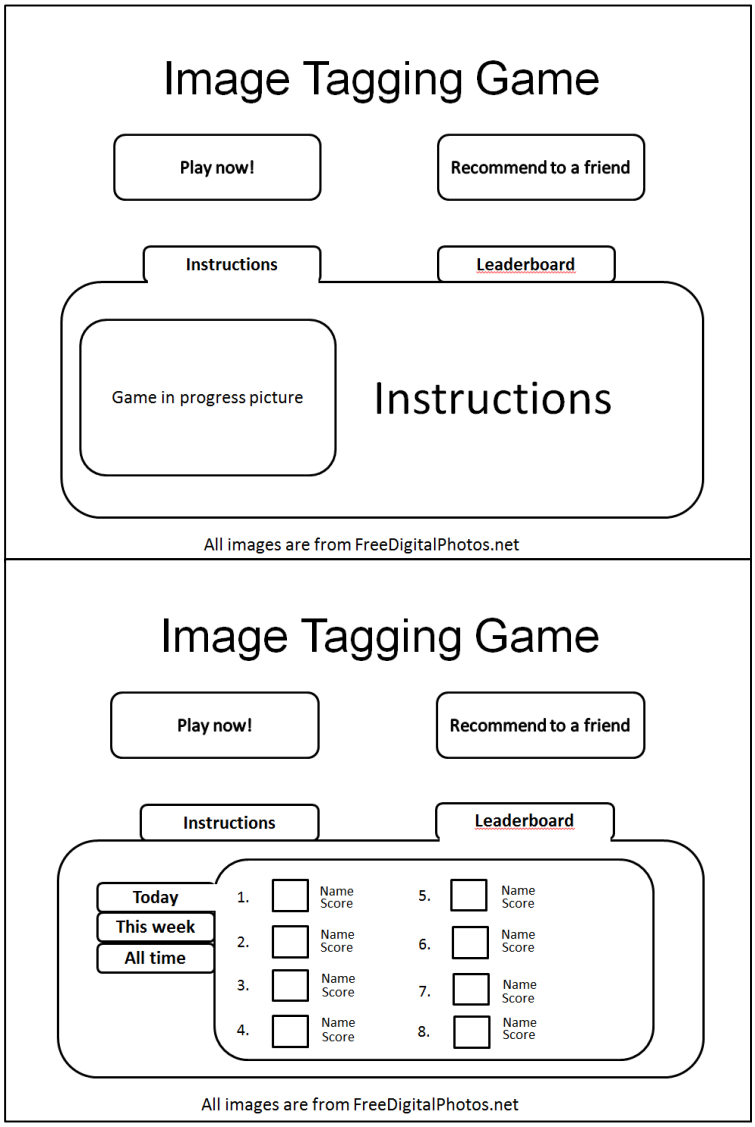


Figure 6: The 'final' game design from the Interim Report

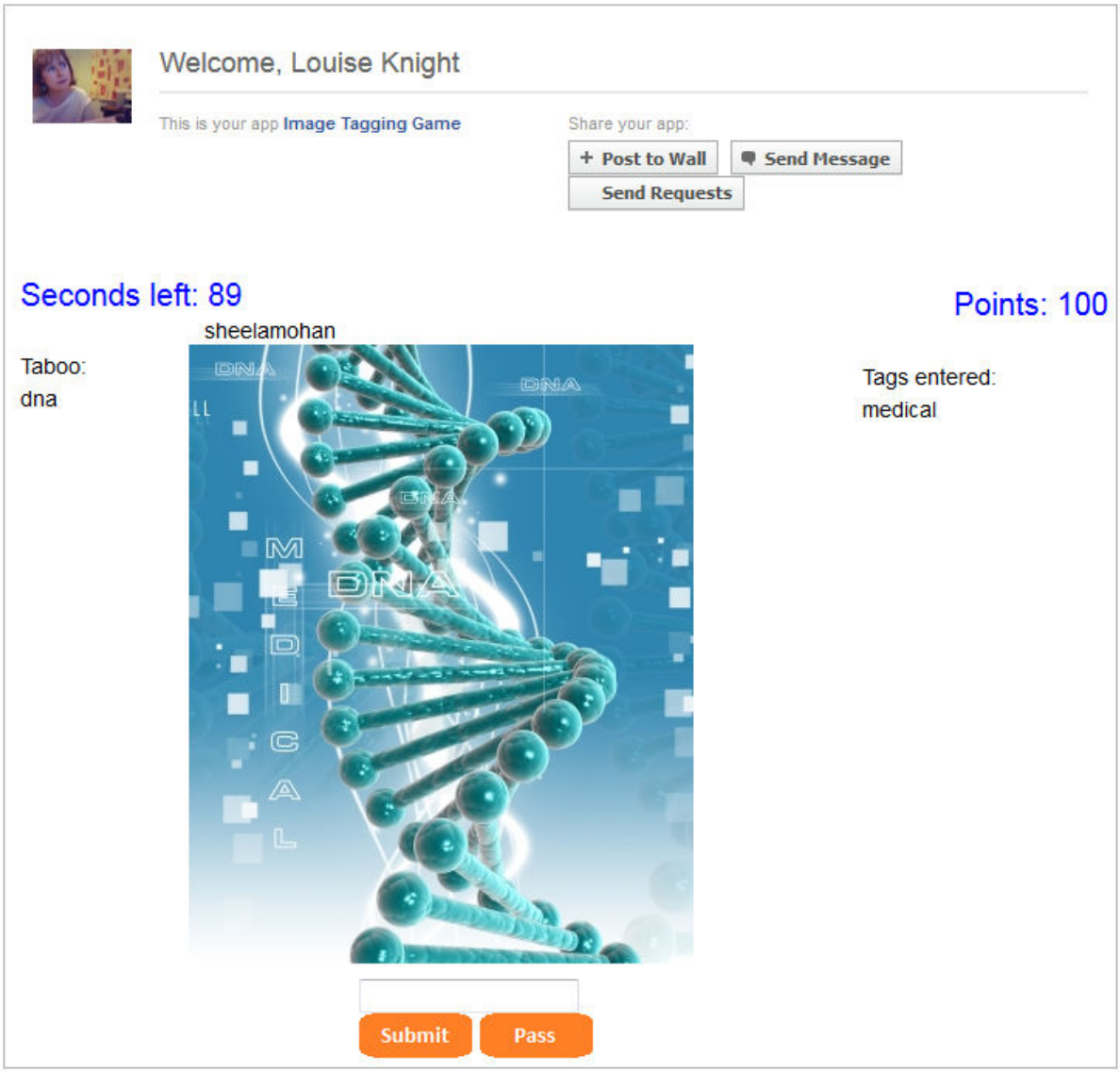


Figure 7: The game design as it was implemented

List of users:
1415572679 1512439047
You are now connected to 1415572679
25 23 fruit 8

Figure 8: Debugging information found at the bottom of the app; the Facebook IDs of the users logged into the game is shown, as well as who the current user is paired up with. Also shown is a list of image IDs alongside the tags agreed on for each image (if two image IDs are shown together, as above, it means that one of the users passed on the first image)

Implementation

Here we shall look at how the main functions of the system were implemented; some such as recommending the game to a friend and authorising the app were already implemented in the sample app provided when it was created on Facebook with Heroku, and so including code snippets of these parts is unnecessary as this was not our own work. As such we shall simply take the 'Play a game' function and break it down, going through the processes a user would go through when playing the game themselves.

Note: The main functionality of the game is in two files, *web.js*, the 'server-side' code, and *index.ejs*, the 'client-side' code. (Appendix A gives the full source code of these files.) As such, we shall distinguish in each of the code snippets where the code can be found. Also, sometimes code will not be shown if we feel doing so is unnecessary in the context of what we are explaining. Where we have omitted such code, we shall use a comment to briefly explain what the omitted code does (unless that code is simply console output for debugging purposes).

The main function used for message passing in *Node.js*, *emit*, can be used in two ways: firstly, with a socket ID, for example sockets[*user*].emit(*data*), where *user* is the socket ID for a user and *data* is the data we wish to send to that user, and can be thought of as sending a point-to-point message to a specific user. This is used to send data from the server to a specific client. The second (and more general) way is by not supplying a socket ID; for example, as in socket.emit(*data*); this is used to send data from the client to the server (of which there is only one).

The data stored for each user is as follows:

- *currentUsers*: a list of the Facebook IDs of all of the users currently logged in to the game
- *sockets*: at the index of each user's Facebook ID, store a socket associated with that user. This allows us to send messages to each specific user.
- *socketToFB*: at the index of each socket ID, store the Facebook ID associated with that socket
- *partners*: at the index of each user's Facebook ID, store the Facebook ID of that user's partner
- *noOfUsers*: simply the number of users logged in to the game
- *guesses:* at the index of each user's Facebook ID, store an array of the tags entered by that user for the image currently being viewed
- *imageIDs:* at the index of each user's Facebook ID, store the ID of the image that the user is currently viewing
- *imageIDsSeen:* at the index of each user's Facebook ID, store an array of IDs for the images already seen in this game

Starting the game

When the user first accesses the app's page, they shall be confronted with a simple interface with a 'Play' button. Clicking this button calls the startAGame function, seen below.

index.ejs

```
function startAGame() {
	FB.api('/me', function(response) {
		fbid = response.id;
		socket.emit('name', fbid);
	});
	// Tell the user to wait a while, hide/show appropriate components of the page
}
```

This function uses the *JavaScript* Facebook SDK (9) in order to find the user's Facebook ID, which is unique for each Facebook user. The SDK can also be used for: providing for logging in using a user's Facebook credentials, allowing users of the app to send messages and requests using Facebook Dialogs, and other similar features. We have chosen to identify each user by their Facebook ID simply because this will not change between browser sessions, and can be found easily using this call, and hence it is useful for associating scores achieved to individual players. We emit this ID with the event *name*. This is picked up on the server side using socket.on('name', ...); as below:

web.js

```
socket.on('name', function(data) {
	if ((currentUsers.indexOf(data)) != -1) {
		socket.emit('already here');
	} else {
		// Store a link to this user's socket using their Facebook ID, increment the
		// number of users currently online
		if (((noOfUsers % 2) == 0) && (noOfUsers != 0)) {
			// Store who each user's partner is
			var r = getRandomInt(1, numberOfImages);
			// Store r as the ID currently being looked at, and add it to a list of
			// images already seen, to avoid duplication later
			sockets[user1].emit('r', r);
			sockets[user2].emit('r', r);
			getTabooWords(r, user1, user2);
			pg.connect(process.env.DATABASE_URL, function(err, client) {
				var query = client.query('SELECT creator FROM images
WHERE image_id = ' + r);
				query.on('row', function(row) {
					var result = JSON.stringify(row);
					result = result.substring(12, result.length - 2);
					sockets[user1].emit('creator', result);
					sockets[user2].emit('creator', result);
				});
			});
			sockets[user1].emit('start timer');
			sockets[user2].emit('start timer');
		}
		guesses[data] = new Array();
	}
});
```

Here we first check to see if we have already stored a link to this person's socket in the *currentUsers* array, if so, this means that the user has probably opened the game in a browser window, while they were still playing the game in another window. In this case, in order to stop errors occurring due to pairing up the same user with more than one partner, we simply tell them that they have already logged into the game, and do nothing further. Otherwise, we wait until we have an even number of users logged into the game, and then we connect each user with a partner, storing any other data necessary such as linking their Facebook ID to their socket. Then, we generate a random integer between 1 and the total number of images (30, in our case) inclusive. This integer is emitted to the two users, and we then find the taboo words for that image using the *getTabooWords* function. Finally, we find the creator of the image and send it to both users, as well as a request to start the timers on the webpage (these start at 90 seconds), and create a new array to hold the guesses for the user that just joined, regardless of whether they have a partner yet or not.

The *getTabooWords* function is below:

web.js

```
function getTabooWords(imageID, user1, user2) {
	pg.connect(process.env.DATABASE_URL, function(err, client) {
		client.query('SELECT tag_text FROM tags, taboo WHERE tags.tag_id=taboo.tag_id AND image_id=' + imageID + ';', function(err, result) {
			sockets[user1].emit('taboo words', result);
			sockets[user2].emit('taboo words', result);
		});
	});
}
```

Having connected to the PostgreSQL database, we use the query to link up the *tags* and *taboo* tables to find if there are any taboo words for this image, and what they are, and we send them to the clients.

Entering a tag

Now we are at the point where we have found, and sent, to each user in a pair: the image ID, the creator for that image, and the taboo words for that image – the client takes this information and uses it to present to the user the image, creator, and taboo words. Now, when a user enters a tag, we emit this tag to the server, where we firstly work out whether the tag just entered is a taboo tag. This ensures that we do no further processing to the tag entered if it is taboo, and process it if it is not taboo:

web.js

```
var taboo = false;
client.query('SELECT tag_text FROM tags, taboo WHERE tags.tag_id=taboo.tag_id AND image_id=' + imageID + ';', function(err, tag) {
	if (tag.rows[0]) {
		for (var i = 0; i < tag.rows.length; i++) {
			if (tag.rows[i].tag_text == data) {
				taboo = true;
			}
		}
		if (taboo) {
			sockets[user].emit('is a taboo');
		} else {
			dealWithTags(user, partner, imageID, data);
		}
	} else {
		dealWithTags(user, partner, imageID, data);
	}
});
```

Using the same query as we did before, we retrieve the taboo words for this image and establish whether the tag entered is taboo (using the first 'for' loop – the *data* variable holds the tag just entered). If so, we emit the event 'is a taboo' to the user (which then fires an alert to tell the user that they have entered a taboo word). If not (or if there are no taboo words for this image at all), we use the *dealWithTags* function to do the rest of the tag processing necessary.

The *dealWithTags* function firstly checks whether there is a match between the tag entered and the list of all tags entered by the user's partner for this image. The matching is exact matching, which means that if, for example, one user entered the Americanised spelling of a word and the other the British spelling of the same word (e.g. 'color' and 'colour'), then the system would not consider this a match. Using exact matching could also create problems if the capitalisation for words is different, or there are typos in a word, etc. These problems could be rectified using a spell-checking API, such as IBM's Jazzy (10).

If there is a match, then one of three things will happen:

1 We have not seen the tag before (for this image or any other), so:
	1.1 Add to the *tags* table
	1.2 Add to the *non_taboo* table for this image, with *agrees_left*=5
2 We have seen the tag before, but not for this image
	2.1 Add to the *non_taboo* table for this image, with *agrees_left*=5
3 We have seen the tag before, and for this image
	3.1 Select the *agrees_left* for this tag and image, and decrement by 1
	3.2 If *agrees_left* == 0, remove this tag from *non_taboo* and insert into *taboo*
	3.3 Else, update *agrees_left* with new value

This functionality is realised in the following code (console.*log* statements for debugging have been removed for clarity):

web.js

For option 1:

```
client.query('INSERT INTO tags (tag_text) VALUES (\'' + data + '\');');
var tagID = client.query('SELECT tag_id FROM tags WHERE tag_text=\'' + data + '\';', function(err) {
	if (err) console.log(err);
});
tagID.on('row', function(row) {
	client.query('INSERT INTO non_taboo VALUES (' + imageID + ', ' + JSON.stringify(row.tag_id) + ', 5);');
	client.query('SELECT * FROM non_taboo WHERE tag_id=' + row.tag_id + ';', function(err, result) {
	});
});
```

For options 2 and 3:

```
client.query('SELECT * FROM non_taboo WHERE image_id=' + imageID + ' AND tag_id=' + result.rows[0].tag_id + ';', function(err, tagID) {

	if (!tagID.rows[0]) { // Option 2

		client.query('INSERT INTO non_taboo VALUES (' + imageID + ', ' + result.rows[0].tag_id + ', 5);');

	} else { // Option 3

		client.query('SELECT agrees_left FROM non_taboo WHERE image_id=' + imageID + ' AND tag_id=' + result.rows[0].tag_id + ';', function(err, agrees) {

			var new_value = agrees.rows[0].agrees_left - 1;

			if (new_value == 0) {
				client.query('INSERT INTO taboo VALUES (' + imageID + ', ' + result.rows[0].tag_id + ');');

				client.query('DELETE FROM non_taboo WHERE image_id=' + imageID + ' AND tag_id=' + result.rows[0].tag_id + ';');

			} else {
				client.query('UPDATE non_taboo SET agrees_left=' + new_value + ' WHERE image_id=' + imageID + ' AND tag_id=' + result.rows[0].tag_id + ';', function(err) {…});
			}
		});
	}
});
```

By this point we have done all the tag processing necessary, so we simply show the next image. To do so, we set each user's array of tags entered to be empty, generate a new image ID, making sure this pair of users has not seen it before, and then set the image ID currently being looked at by each user in this pair to the new ID, and push it onto the list of IDs seen. We emit the ID and creator of the image, as well as the taboo words for this image, which we then display to the user, as before.

Pass button

If one of the users in a pair feels that matching on a tag for the image being viewed is too difficult, they may choose to click the 'Pass' button. On doing so, we tell the user's partner that their partner has passed, and ask them to confirm to pass on this image, before the next image is shown. This avoids the other user becoming confused as to why the image shown has changed.

Firstly, the user clicking the 'Pass' button causes the emitting of the 'pass' event, which is subsequently detected on the server-side:

index.ejs
```
function passOnThisImage() {
	socket.emit('pass');
}
```

web.js
```
socket.on('pass', function() {
		var user = socketToFB[socket.id];
		var partner = partners[user];
		sockets[partner].emit('passed');
});
```

As you can see, in *web.js* (on the server side), we then emit the 'passed' event to the partner of this user, which is then detected on that user's client-side:

index.ejs
```
socket.on('passed', function() {
	document.getElementById("passed").innerHTML = "Your partner has passed on this image!";
	document.getElementById("confirmPass").style.visibility = "visible";
});
```

The user is given a message about the partner passing, and a 'Confirm' button is shown. Once the user clicks this, the following function is fired:

index.ejs
```
function confirmPassOnThisImage() {
	document.getElementById("confirmPass").style.visibility = "hidden";
	document.getElementById("passed").innerHTML = "";
	socket.emit('confirm');
}
```

As you can see, a 'confirm' event is emitted, which is then detected on the server:

web.js

```
socket.on('confirm', function() {
        var user = socketToFB[socket.id];
        var partner = partners[user];
        showNextImage(user, partner);
});
```

The *showNextImage* function is called, which does all the work to show the next image, as described previously (i.e. generate a new image ID, making sure we have not seen it before, etc.).

The game ends

A counter is shown when the game is being played, and it counts down from 90 to 0. When we reach 0, we emit an event to let the server know that we've finished the game:

index.ejs

```
function timer() {
    count -= 1;
    if (count == 0) {
       socket.emit('time up', points);
       clearInterval(counter);
    }
    document.getElementById("time").innerHTML = "Seconds left: " + count;
}
```

Once this has been detected on the server side, we either:

- Insert the score attained as the maximum score and cumulative score, if this user has not played before, or
- Update the maximum score and cumulative score as appropriate, if they have played the game before

The code to deal with these events is below:

web.js

```
client.query('SELECT * FROM users WHERE user_id=' + user + ';', function(err, result) {
	if (err) {
		console.log(err);
	} else {
		if (!result.rows[0]) { // User has not played before
			client.query('INSERT INTO users VALUES (' + user + ', ' + data + ', '
+ data + ');');
		} else { // User has played before
			client.query('UPDATE users SET total_score=total_score+' + data + '
WHERE user_id=' + user + ';');

			var query = client.query('SELECT score FROM users WHERE
user_id=' + user + ';');

			query.on('row', function(row) {
				var score = JSON.stringify(row.score);
				if (data > score) { // new score larger than old maximum
					client.query('UPDATE users SET score=' + data + '
WHERE user_id=' + user + ';');
				}
			});
		}
	}
});
```

Having done this, we now clear up the data structures used during the game, deleting them as appropriate.

If one of the users has closed the game window before time is up, we need to tell their partner this and abort the game. We detect the action of a user closing the game window as so:

index.ejs

```
window.onbeforeunload = closing;

function closing() {
	socket.emit('closing', fbid);
}
```

The emitting of the 'closing' event is picked up on the server-side, and it is here we firstly clear up the data structures used as we would if the game finished normally, and then alert the partner of this user that they have left the game.

Problems encountered

Here we shall discuss the problems encountered during the course of the project, in the order that they were encountered.

During the design stage of the project, the mistake was made of committing to using some technology before thoroughly researching its uses and benefits. This occurred largely because

of a lack of knowledge of developing Facebook applications and also not taking into account the various characteristics that would be desirable of the language we would use. Wanting to use 'familiar' languages, we initially researched technologies that Facebook supported with dedicated SDKs rather than first specifying what we were looking for in a language; properties such as being good for real-time, multiplayer applications, for example. In the space of weeks the choice of technologies to use moved from *PHP* to *Java* with web sockets, to *Node.js*. It should be noted, however, that this 'problem' worked out well in the end in that through the research and exploration of various languages that could be used, we managed to find the most appropriate one, which made the development of the app fairly straightforward.

Next, once we had decided to use *Node.js*, we tried too quickly to start developing when more time should have been spent on first learning the concepts related to developing in *Node.js*; because of this, the Christmas vacation was spent doing what should have been done before, learning about how *Node.js* works in conjunction with *Socket.IO*, and it was only at the start of the second semester that the concepts were understood at a level that made it apparent that it would actually be fairly easy to implement the system. More time should have been spent doing this at the beginning of the project.

Too much time was spent developing the design for the system, in particular the interface design, when this was not the real focus of the project. The reason for having done this is not having done a large individual project before, and so in order to move forward with the project, we were simply trying to borrow ideas from the only other substantial project completed before this point: the group project module in Year 2 (which devoted more time to interface design, which was more appropriate for that project than this one). The time spent on design could have been spent on implementation instead. In the future we should spend more time considering the languages/technologies to use and less on design (unless it is a particularly large and complex project).

A problem which surfaced towards the end of the project is that there was a change in some code that the app depended on, causing the app to completely fail. It took approximately a week to locate the source of the problem and resolve it. This could not have been anticipated and meant the loss of critical development time, especially considering this problem surfaced towards the end of the project. There was a lack of documentation as to how to fix this problem; we relied upon other developers' recommendations, hence the disproportionately large amount of time dedicated to solving a problem which only required a couple of lines of code to be changed in order to fix.

Although there seems to be a lot of negativity surrounding the problems which arose, the 'problems' which did occur during this project were, at least in part, inevitable due to our inexperience in developing Facebook applications, and it is through encountering these issues that we have learned a lot about not only *Node.js* and Facebook app development, but in how to solve problems and plan projects in general.

We have discussed all pertinent information concerning the implementation of our system; we have taken relevant pieces of code from the system and described how they work together in order to provide the different functionalities. We have also discussed problems encountered during this stage of the project and how they were dealt with.

Results and Evaluation

Here we shall discuss the results of the project, as well as evaluate the project against the aims we had decided upon at the beginning of the project. We shall also evaluate our choices of method and technology in implementing the system.

Results of Acceptance Testing

Mostly due to time constraints, testing simply consisted of sitting down and making sure all of the various pieces of functionality worked. We were unable to obtain testers for the game, but were able to play it with a couple of pairs of users, and this testing was wholly successful. Also, time was too limited to use the ReCal3 (11) tool to calculate the inter-rater reliability in order to record the amount of agreement between the users playing the game in tagging images. We are very confident that the game works for small numbers of users, but more rigorous testing is required to establish its scalability to larger numbers of users. We also know that any tags entered into the system are handled correctly, at least for small numbers of users. Appendix B gives an example of how we carried out this testing.

Evaluation of the project as a whole against project aims

Here we shall examine each aim in turn and evaluate how this aim has been met (or not), with a discussion of why if appropriate. The bullet-pointed aims are taken directly from the Initial Plan for this project.

Core aims

- Create a Facebook game which allows for:
    - the tagging of images in [an ESP Game-manner] (these images [shall be] manually placed in the game, although [we shall] consider allowing for Facebook images to be included, for example
    - timed games (for example, 2 minutes)
    - taboo words, which means that once a label has been successfully assigned (having passed the "good label threshold"), it shall appear on a list of taboo words that the players cannot use to assign to that particular image

This has been met during the course of the project – we have created a Facebook game that allows for the tagging of images in 'an ESP Game-manner'; by that we mean that we pair up users with a partner, they are shown an image, and then they enter tags to describe that image. If they match, they both get points. The images were manually placed; they were downloaded from a royalty-free image website (FreeDigitalImages.net) and were then hosted on an external website (Cloudinary) by manually uploading the images. As far as allowing for Facebook images is concerned, we did not have enough time to consider this possible application of the game. Of course, if we were to implement this feature, the privacy of users would have to be considered (if someone signed up to our game, we could ask them if they would allow their images to be used in the game, or even to allow them to select specific images they would allow to be used).

The game is a timed game.

The concept of taboo words is firmly embedded in the game. As an example, we used a ‘good label threshold’ of 5 for our game, so a tag needs to be agreed on 5 times for a particular image before it becomes a 'proper' tag and is considered reliable enough to be considered a tag. Once a tag has become a taboo word, we place it in a list of taboo words shown the next time that particular image is shown. If a user tries to enter that word as a tag once it is in the list of taboo words for that image, they are told that they cannot do so because it is taboo.

- Allow users to play against friends (but need to think about the possibility of cheating)

It is entirely possible to play against friends in this game, but during the course of a game, the user is never told who their partner is, eliminating the possibility of cheating. If the identity of a user’s partner was seen, then if the user were friends with that person, they would be able to start a Facebook conversation with them and, for example, agree to always enter 'a' as a tag, in order to exploit the tag-matching process and win points. (Although, if that did happen, this is why we have the concept of a ‘good label threshold’, so if such a spam tag were entered, it would not automatically become a 'proper' tag for that image.)

- Implement a facility which means that if only one person is playing at a time, they can essentially “play” against someone who has already played before via playing through someone's guesses

This feature was the only core feature not implemented due to the time constraints mentioned previously. What would need to be done to implement this is given in the ‘Future Work’ section.

- Simple posting to user's Timeline to recommend the game to a friend

This kind of feature was implemented, along with a couple of others. For example, we can send a Facebook message to a friend with a link to the game itself, and we can also send requests to friends which would appear in the App Centre 'Requests' section. In addition, we can post to our own Timeline about the game with a link to it.

Optional aims

None of the optional aims specified in the original plan were implemented; we shall address the changes and additions that would need to be made to the game in order to implement these features in the ‘Future Work’ section of this report.

How the project was conducted

We feel that the methodology chosen for the project (i.e. using the agile principles as a guide) was relevant in ensuring that the focus was on the core requirements, although we must admit that the principles were given little thought once development had commenced. That being said, the principles picked out in the Interim Report (especially concerning frequent delivery and maximising simplicity) were adhered to automatically when actually implementing the system. For example, during the implementation, we looked for easier ways of doing things, trying to minimise the amount of code written, and attempted to implement the system and its functionalities as quickly as possible so that we always had a working system.

On the other hand, the work plan devised for the project was overambitious and we knew when implementation began properly in the second semester that it would be unlikely that we would be able to implement all of the extra features proposed. That being said, it is felt that adequate time was spent not only implementing the system but also writing the report.

Technical aspects

Node.js was chosen as the main programming language for the system, with *Socket.IO* for enabling communication between the various components. This meant that *JavaScript* was used for both the client-side and server-side code. This was a very good choice in that we already had experience of working with *JavaScript*, and, combined with the use of Heroku for hosting the app, meant that it was effortless to set up the app (Facebook provides a sample app with some basic functionality provided in whichever language you choose if you select Heroku for hosting). Once progress had been made in understanding the *Node.js*/*Socket.IO* languages and how they work, implementation was fairly straight-forward in that, when you want to send data between the client- and server-sides of the code, you simply need an *emit* function sent to the socket you want to send to.

Heroku was a good choice for our app hosting needs, again in its easy integration with Facebook and the fact that it supported both *Node.js* and *Socket.IO*, but of course, there are limitations with choosing such a platform (as there are with choosing to host your code on any cloud computing platform). For example, issues arose during the implementation phase which could not have been foreseen, such as a change being made to code on Heroku's side causing our app to fail completely, which meant having to contact other developers to find a suitable workaround. But, again, of course, such an error could arise on any website where you are hosting your code, as you are depending on others' resources.

PostgreSQL was a good choice for our database for its integration with Heroku and its relatively easy-to-use API for *Node.js*. Also, a GUI program for accessing PostgreSQL databases was easy to find and use (pgAdmin III (12)). Such a GUI program was necessary so that the database could be viewed before and after changes were made to the tags stored, for example, which allowed us to check that tags were being handled successfully by the app, and therefore this program proved to be a crucial part of our testing strategy.

As mentioned previously, Cloudinary was used to store the images, and this meant that they could be accessed easily through the use of URLs (if we were to receive the image ID in the *data* variable, we could use the URL "http://res.cloudinary.com/hd93ufv7u/image/upload/" + *data* + ".jpg" to refer to the image associated with that ID).

We have discussed the results obtained through the project, and how much we can rely on them. We have also described our choices of methodology and language use throughout the project, and how these have impacted on the project itself.

Future Work

There were some ideas that we did not have time to implement, only one of which was a core requirement. Here we shall discuss not only what would need to be done in order to implement these unrealised requirements, but also some smaller points which could be addressed in the future in order to improve the app further.

The only core requirement which was not implemented was to allow a user to 'play' using previously-entered guesses, so that, if only one user was using the system, for example, or an odd number were, then they would not have to miss out on playing the game. To make this feature possible, we would need to collect tags as they are being entered and play them back as appropriate, not only when only one user is playing, but also if a user's partner leaves before the game ends. It should be noted, though, that we would have to make the game 'live' for quite a while in order to gather tags for each image (ideally, tags for most of the images in the system).

Now we shall examine the optional aims in turn and consider the changes that would need to be made to the app to support these.

Leader board of high scores (possibly including "all time", "today", etc.) (and hence high score posts to the user's Timeline)

We are already storing the user's maximum score achieved; this could be used to add a leader board to the front page listing the users with the highest scores; in addition, leader board tables could be made in the database in order to address the different time scales ('all time', 'today', etc.) and make displaying the top scores for each time period easier. High score posts could be made using a similar dialog to that used to currently post to one's Timeline.

Ranks applied to each user once they have passed a certain number of points (for example, Level 1, 2, etc.)

We are already storing the accumulated scores for each user, so it would simply be a case of adding a rule to the game such as 'After every game is complete, add the points for that game to the points already accumulated for that user; if this total is over some threshold, the user has acquired rank X'. We could then give the user the option to post to their Timeline each time they acquire a new rank.

The ability to review words assigned to each image at the end of the game

This would take a little more effort than the other optional aims to implement; we would have to store the tags entered for an entire game for both of the users, and then print this to the screen at the end of a game alongside the images shown; we are already recording the images 'seen' in a particular game in order to eliminate duplication, as well as the tags entered by each user for each image, but we would need to change the way these are being stored so that we can keep all of this data throughout a game, rather than throwing it away every time we change the image.

There are other changes that could be made to the app that would further improve it. One of these is to make the app work in Internet Explorer. As it is, the app doesn't work in this browser, but it does work in Mozilla Firefox and Google Chrome.

Another change that could be made is to improve the interface of the app. Not so much time was devoted to making the app aesthetically pleasing in the implementation stage of the project, and it could be improved in the future.

Also, some elements of the app have, in hindsight, not been implemented as efficiently as they could have. For example, the current way of checking that the tag just entered is not a taboo word is really not efficient; we need to send an alert to the server, which makes calls to the database to retrieve the list of taboo tags for this image, and then it checks whether the tag entered is one of them. Instead, this processing should been done on the client-side. To do this, we could store the array of taboo words returned by the call to the database on the client, and whenever a tag is entered, check the tag against the array stored for each user.

Another thing which could be improved is that we are currently storing some data 'twice' when it is not necessary. For example, we are storing the image ID that each user is looking at currently, when really we need only do so for each *pair* of users, since the image they would be looking at would be the same. To only store it once, we could utilise the *partners* array. This means that to access, for example, the image ID being looked at by a user, we could first try to index into *imageIDs* using the Facebook ID of that user, and if this does not work, find their partner and use the partner's ID. This, of course, means a couple more calculations are necessary, but should, as the number of users in the game increases, mean the storage space needed for this data is cut in half.

Something specified in the Interim Report was to use a tool such as ReCal3 in order to find the inter-rater agreement on tags found in the game. This wasn't done due to time constraints, so after the app has been live for a while, this tool could be used to assess the tags collected in the game.

Another feature which was also mentioned was allowing for the tagging of Facebook images. To do so would, inevitably, mean the consideration of some privacy issues. To work around this, we could ask the user playing the game if they wish to use some of their own images in the game, and give them the option of uploading them should they wish to do so. Doing this would not allow us to control the content of the images though, and there is also the possibility of users 'spamming' the system with images, so there could be a system where points accumulated during the game could amount to a level of trust associated with each user. Users with very low trust would not be allowed to upload their own images, ones with medium levels might be able to upload them once they have been screened for appropriateness, and users with very high scores (and therefore trust) would not need their images to be screened. Abuses of the system would cause points to be deducted, lowering our trust in that user, and users who upload good quality images would be rewarded for doing so.

We have described how we would improve the project, given more time, and discussed the changes that would need to be made in order to do so.

Conclusions

The aims of the project were to make an ESP-style game, and implement it as a Facebook application, taking advantage of the fact that, unlike the ESP Game on the GWAP website, its presence on a social networking website would encourage users to invite their friends to play the game, further encouraging more people to play the game and allowing for a wider range of tags to be gathered.

We have implemented this game so that it is functional for equal numbers of users, and incorporates the ideas of a 'good label threshold', taboo words, etc. Also, a scoring system has been implemented in order to encourage healthy competition among players, and other features, such as the ability to recommend the game through sending a message to a friend or inviting them through a request means that, as more and more people play the game, we will be able to gather more tags.

The testing of this game has established that the game works for multiple users. However, as mentioned, further research of the tags gathered would be interesting in allowing us to compare the tags collected by multiple users, but this is reserved as future work due to the time constraints encountered.

Some difficulties arose during the project, including some that we could not plan for (for example, our app crashing due to a change made to code that our app depended on), and some that were instrumental in allowing the project to progress further (for example, choosing the right language to use). We have explained these problems and what we could do differently in the future.

We have also clarified the design of the system in terms of its functionality and database design, etc., and have analysed the results we have found through the project.

Reflection on Learning

Here we shall reflect on the skills learnt throughout the project, including technical and project management skills.

Some of the more obvious skills that have been developed during this project are our technical skills. Although already having some experience of web programming in the sense that we have previously completed a project using web technologies such as *PHP* and *JavaScript*, we had not previously completed any programming for real-time applications. Also, we had never developed Facebook applications previously, which meant that initial research into the technologies available was necessary to find out about the various languages that could be used, and then selecting one based on its functionality and also similarity to previous languages learnt, in order to make the development more straightforward. It is through this project that we have learnt the basic concepts behind, and learnt to program in, *Node.js* and *Socket.IO*, while also learning how to utilise the Facebook API to collect basic information about a user, such as their Facebook ID.

We learned a lot about software development and project management in general in this project, and this was helped along by the various hurdles that were cleared throughout the course of the project. For example, having successfully researched into the various technologies available for Facebook app development, we then needed to break the project down into different pieces of functionality. This really stretched our ability to visualise a complex system and then think about how each component interacts with and influences every other component.

Through the project we have learned that initial plans will often change, and assumptions about work to be done will be tested. For example, initially we planned to have implementation finished by week 5 of the spring semester, but we did not anticipate the unavoidable issues that arose. Because of this, we could not implement every single feature, but because of the way we implemented the system and how we chose to make sure the absolute essentials were covered first, we were still able to cover most of the use cases that would surface throughout a game (for example, when a user leaves the game, we alert their partner to this). On the other hand, some things that were planned to take a long time to implement did not take nearly as long once implementation had begun. In addition, we can say that we have learned not to overcomplicate projects - this project was about tackling a problem, which on further inspection was not too difficult once we had broken down the various pieces of functionality into manageable chunks.

Our problem-solving abilities were also improved through this project, through not only fixing bugs, but also finding out how to implement some feature by researching into what needed to be done and trying to find a piece of software that would plug some gap (for example, finding a GUI program to see how the data in the database changed over time).

Another thing learned is that it is always a good idea to keep a diary of project work throughout a project, as it aids in the writing of a report later on. During the implementation all of the decisions made are fresh in the mind, and so writing them down as you are working adds clarity to the report. In connection to this, our written communication skills have been refined as a result of this project - before this, our written work has not needed to be so strictly-defined in terms of report sections, vocabulary used, etc. This project has improved our ability to write academic reports.

More generally, we have become much more independent in our research through this project. This being the first project we had done individually, at first it seemed almost unmanageable; however, once it had been broken down into tasks that were more easily understood, it was readily apparent that the main task in this project was not in the implementation, but was in staying motivated in order to complete it alongside other commitments.

Appendix A

web.js

```
var async   = require('async');
var express = require('express');
var util    = require('util');
var pg = require('pg');
var socket = require('socket.io');
var http = require('http');
var faceplate = require('faceplate');

// create an express webserver
var app = express.createServer(
  express.logger(),
  express.static(__dirname + '/public'),
  express.bodyParser(),
  express.cookieParser(),
  // set this to a secret value to encrypt session cookies
  express.session({ secret: process.env.SESSION_SECRET || 'secret123' }),
  require('faceplate').middleware({
    app_id: process.env.FACEBOOK_APP_ID,
    secret: process.env.FACEBOOK_SECRET,
    scope:  'user_likes,user_photos,user_photo_video_tags'
  })
);

var io = socket.listen(app);
io.configure(function () {
  io.set("transports", ["xhr-polling"]);
});

// listen to the PORT given to us in the environment
var port = process.env.PORT || 3000;

app.listen(port, function() {
  console.log("Listening on " + port);
});

app.dynamicHelpers({
  'host': function(req, res) {
    return req.headers['host'];
  },
  'scheme': function(req, res) {
    req.headers['x-forwarded-proto'] || 'http'
  },
  'url': function(req, res) {
    return function(path) {
      return app.dynamicViewHelpers.scheme(req, res) +
app.dynamicViewHelpers.url_no_scheme(path);
```

```
    }
  },
  'url_no_scheme': function(req, res) {
    return function(path) {
      return '://' + app.dynamicViewHelpers.host(req, res) + path;
    }
  },
});

function render_page(req, res) {
  req.facebook.app(function(app) {
    req.facebook.me(function(user) {
      res.render('index.ejs', {
        layout:    false,
        req:       req,
        app:       app,
        user:      user
      });
    });
  });
}

function handle_facebook_request(req, res) {

  // if the user is logged in
  if (req.facebook.token) {

          async.parallel([
      function(cb) {
        // query 4 friends and send them to the socket for this socket id
        req.facebook.get('/me/friends', { limit: 4 }, function(friends) {
          req.friends = friends;
          cb();
        });
      },
      function(cb) {
        // query 16 photos and send them to the socket for this socket id
        req.facebook.get('/me/photos', { limit: 16 }, function(photos) {
          req.photos = photos;
          cb();
        });
      },
      function(cb) {
        // query 4 likes and send them to the socket for this socket id
        req.facebook.get('/me/likes', { limit: 4 }, function(likes) {
          req.likes = likes;
          cb();
        });
      },
      function(cb) {
```

```
      // use fql to get a list of my friends that are using this app
      req.facebook.fql('SELECT uid, name, is_app_user, pic_square FROM user WHERE uid
in (SELECT uid2 FROM friend WHERE uid1 = me()) AND is_app_user = 1',
function(result) {
        req.friends_using_app = result;
        cb();
      });
    }
  ], function() {
    render_page(req, res);
  });

 } else {
  render_page(req, res);
 }
}

app.get('/', handle_facebook_request);
app.post('/', handle_facebook_request);

// UP UNTIL THIS POINT IS CODE PROVIDED IN THE SAMPLE APP

// Below from https://developer.mozilla.org/en-
US/docs/JavaScript/Reference/Global_Objects/Math/random
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

var currentUsers = new Array(); // holds a list of the connected users' Facebook IDs
var sockets = {}; // holds a socket to each of the connected users
var socketToFB = {}; // can be used to map a user's socket ID to their Facebook ID
var partners = {}; // tells us, for each user, who their partner is
var noOfUsers = 0;
var guesses = new Array();
var imageIDs = new Array(); // for every user, which image are they looking at?
var imageIDsSeen = new Array(); // for every user, which images have they already seen in
this round?
var numberOfImages = 30;

function getTabooWords(imageID, user1, user2) {
        pg.connect(process.env.DATABASE_URL, function(err, client) {
                client.query('SELECT tag_text FROM tags, taboo WHERE
tags.tag_id=taboo.tag_id AND image_id=' + imageID + ';', function(err, result) {
                        sockets[user1].emit('taboo words', result);
                        sockets[user2].emit('taboo words', result);
                });
        });
}

function dealWithTags(user, partner, imageID, data) {
```

```
	pg.connect(process.env.DATABASE_URL, function(err, client) {
		console.log('Client ID: ', user, ' and tag: ', data);
		guesses[user].push(data);
		console.log('List of tags entered by this client: ');
		for (var i = 0; i < guesses[user].length; i++) {
			console.log(guesses[user][i] + " ");
		}
		// Check for a match
		for (var i = 0; i < guesses[partner].length; i++) {
			if (guesses[partner][i] == data) { // There is a match
				console.log("Match! between ", guesses[partner][i], " and ",
data);
				// check if these users have entries in the database; if they do,
update their score
				sockets[user].emit('matched', data);
				sockets[partner].emit('matched', data);
				// 1. check for tag in 'tags'
				// 2. if not in 'tags', add to 'tags' and put in non-taboo (if not in
'tags', we know it won't be a tag for that image)
				// 3. if in 'tags', check for that tag for that image
				// 4. if already a tag, decrement agrees_left, and if 0, migrate to
'taboo'
				// 5. if not a tag, add to 'non-taboo' and set agrees_left to some
value (let's say 5 for now)
				client.query('SELECT tag_id FROM tags WHERE tag_text = \"
+ data + '\';', function(err, result) {
					if (err) {
						console.log(err);
					} else {
						if (!result.rows[0]) {
							console.log(data + ': isn\'t in the tag table
yet');
							client.query('INSERT INTO tags
(tag_text) VALUES (\" + data + '\');');
							console.log(data + ': added to the tags
table');
							var tagID = client.query('SELECT tag_id
FROM tags WHERE tag_text=\" + data + '\';', function(err) {
								if (err) console.log(err);
							});
							console.log(data + ': querying for the
tag_id (just generated)');
							tagID.on('row', function(row) {
								client.query('INSERT INTO
non_taboo VALUES (' + imageID + ', ' + JSON.stringify(row.tag_id) + ', 5);');
								console.log(data + ': adding to
non_taboo table with count starting at 5, tag_id=' + JSON.stringify(row.tag_id), " and
imageid = " + imageID);
								client.query('SELECT * FROM
non_taboo WHERE tag_id=' + row.tag_id + ';', function(err, result) {
```

```
                                                                    console.log(data + ': ' +
JSON.stringify(result));
                                                                });
                                                            });
                                                        } else {
                                                            // check in non-taboo
                                                            client.query('SELECT * FROM
non_taboo WHERE image_id=' + imageID + ' AND tag_id=' + result.rows[0].tag_id + ';',
function(err, tagID) {
                                                                if (!tagID.rows[0]) {
                                                                    client.query('INSERT
INTO non_taboo VALUES (' + imageID + ', ' + result.rows[0].tag_id + ', 5);');
                                                                } else {
                                                                    client.query('SELECT
agrees_left FROM non_taboo WHERE image_id=' + imageID + ' AND tag_id=' +
result.rows[0].tag_id + ';', function(err, agrees) {
                                                                        if (err)
console.log(err);
                                                                        var new_value =
agrees.rows[0].agrees_left - 1;
                                                                        if (new_value ==
0) {
                                                                            // take tag
from non_taboo, put in taboo table

        client.query('INSERT INTO taboo VALUES (' + imageID + ', ' +
result.rows[0].tag_id + ');');

        client.query('DELETE FROM non_taboo WHERE image_id=' + imageID + ' AND
tag_id=' + result.rows[0].tag_id + ';');
                                                                        } else {

        client.query('UPDATE non_taboo SET agrees_left=' + new_value + ' WHERE
image_id=' + imageID + ' AND tag_id=' + result.rows[0].tag_id + ';', function(err) {
                                                                            if
(err) console.log(err);
                                                                            });
                                                                        }
                                                                    });
                                                                }
                                                            });
                                                        }
                                                    });
                                                }
                                            }
                                        });
                                        showNextImage(user, partner);
                                    }
                                }
        });
}
```

```
function showNextImage(user, partner) {
	guesses[user].length = 0;
	guesses[partner].length = 0; // this should empty the arrays
	var r = getRandomInt(1, numberOfImages);
	console.log('Generate random int: ' + r);
	while (imageIDsSeen[user].indexOf(r) != -1) { // generate a new image ID as long as
the one generated has already been seen
		r = getRandomInt(1, numberOfImages); // this will ensure each pair of users
always sees a random image
		console.log('Generate random int: ' + r);
	}
	imageIDs[user] = r; // simply stores the ID for each user's Facebook ID
	imageIDs[partner] = r;
	imageIDsSeen[user].push(r);
	imageIDsSeen[partner].push(r);
	sockets[user].emit('r', r);
	sockets[partner].emit('r', r);
	pg.connect(process.env.DATABASE_URL, function(err, client) {
	   var query = client.query('SELECT creator FROM images WHERE image_id = ' +
r);
		query.on('row', function(row) {
		var result = JSON.stringify(row);
		   result = result.substring(12, result.length - 2);
		   sockets[user].emit('creator', result);
		   sockets[partner].emit('creator', result);
		});
	});
	getTabooWords(r, user, partner);
}

io.sockets.on('connection', function(socket) {
	socket.on('name', function(data) {
		if ((currentUsers.indexOf(data)) != -1) {
			socket.emit('already here');
		} else {
			currentUsers[noOfUsers] = data;
			sockets[data] = socket;
			socketToFB[socket.id] = data;
			noOfUsers++;
			console.log('Number of users: ' + noOfUsers);
			console.log('User added: ' + data);
			socket.broadcast.emit('users', currentUsers);
			socket.emit('users', currentUsers);
			if (((noOfUsers % 2) == 0) && (noOfUsers != 0)) { // if we have an
even number of users
				var user1 = currentUsers[noOfUsers - 2];
				var user2 = currentUsers[noOfUsers - 1];
				console.log('user1: ' + user1 + ', user2: ' + user2);
				sockets[user1].emit('partner', user2);
				sockets[user2].emit('partner', user1);
```

```
                                sockets[user1].emit('done it');
                                sockets[user2].emit('done it');
                                partners[user1] = user2;
                                partners[user2] = user1;
                                console.log('user 1\'s partner: ', partners[user1], ' and user 2\'s
partner: ', partners[user2]);
                                var r = getRandomInt(1, numberOfImages);
                                imageIDs[user1] = r; // simply stores the ID for each user's
Facebook ID
                                imageIDs[user2] = r;
                                imageIDsSeen[user1] = new Array();
                                imageIDsSeen[user2] = new Array();
                                imageIDsSeen[user1].push(r);
                                imageIDsSeen[user2].push(r);
                                sockets[user1].emit('r', r);
                                sockets[user2].emit('r', r);
                                getTabooWords(r, user1, user2);
                                pg.connect(process.env.DATABASE_URL, function(err,
client) {
                                    var query = client.query('SELECT creator FROM images
WHERE image_id = ' + r);

                                        query.on('row', function(row) {
                                    var result = JSON.stringify(row);
                                        result = result.substring(12, result.length - 2);
                                        sockets[user1].emit('creator', result);
                                        sockets[user2].emit('creator', result);
                                    });
                            });
                            sockets[user1].emit('start timer');
                            sockets[user2].emit('start timer');
                            }
                            guesses[data] = new Array();
                    }
    });

    socket.on('id', function(data) {
                    var r = getRandomInt(1, numberOfImages);
                    var client = data;
                    imageIDs[client] = r; // simply stores the image ID for each user's Facebook
ID
                    sockets[client].emit('r', r);
                    sockets[partners[client]].emit('r', r);
        pg.connect(process.env.DATABASE_URL, function(err, client) {
                var query = client.query('SELECT creator FROM images WHERE image_id = '
+ r);

                query.on('row', function(row) {
                        var result = JSON.stringify(row);
                        result = result.substring(12, result.length - 2);
```

```
                sockets[client].emit('creator', result);
                sockets[partners[client]].emit('creator', result);
            });
        });
        getTabooWords(r, client, partners[client]);
    });

    socket.on('tag', function(data) {
            var user = socketToFB[socket.id];
            var partner = partners[user];
            var imageID = imageIDs[user];
            pg.connect(process.env.DATABASE_URL, function(err, client) {
                var taboo = false;
                client.query('SELECT tag_text FROM tags, taboo WHERE
tags.tag_id=taboo.tag_id AND image_id=' + imageID + ';', function(err, tag) {
                    if (tag.rows[0]) {
                        for (var i = 0; i < tag.rows.length; i++) {
                            console.log('tag.rows[i].tag_text=' +
tag.rows[i].tag_text + ', tag=' + data);
                            if (tag.rows[i].tag_text == data) {
                                taboo = true;
                            }
                        }
                        if (taboo) {
                            sockets[user].emit('is a taboo');
                        } else {
                            dealWithTags(user, partner, imageID, data);
                    }
                    } else {
                        dealWithTags(user, partner, imageID, data);
                    }
                });
            });
        });

        socket.on('pass', function() {
            var user = socketToFB[socket.id];
            var partner = partners[user];
            sockets[partner].emit('passed');
        });

        socket.on('confirm', function() {
            var user = socketToFB[socket.id];
            var partner = partners[user];
            showNextImage(user, partner);
        });

        socket.on('time up', function(data) {
            var user = socketToFB[socket.id];
            // put score in database for this user, in particular:
```

```
                // if user is in db, update score
                // if not, insert into db
                console.log("Time's up!, user = " + user + ", points = " + data);
                pg.connect(process.env.DATABASE_URL, function(err, client) {
                        client.query('SELECT * FROM users WHERE user_id=' + user + ';',
function(err, result) {
                                if (err) {
                                        console.log(err);
                                } else {
                                        if (!result.rows[0]) {
                                                client.query('INSERT INTO users VALUES (' +
user + ', ' + data + ', ' + data + ');');
                                        } else {
                                                client.query('UPDATE users SET
total_score=total_score+' + data + ' WHERE user_id=' + user + ';');
                                                // get current top score, compare to score just
obtained, put new one in
                                                var query = client.query('SELECT score FROM
users WHERE user_id=' + user + ';');
                                                query.on('row', function(row) {
                                                        var score = JSON.stringify(row.score);
                                                        if (data > score) {
                                                                client.query('UPDATE users SET
score=' + data + ' WHERE user_id=' + user + ';');
                                                        }
                                                });
                                        }
                                }
                        });
                });
                socket.emit('redirect');
                var i = currentUsers.indexOf(user);
                if (i != -1) { // So it won't do this if the user hasn't started a game
                        currentUsers.splice(i, 1);
                        imageIDs.splice(i, 1);
                        delete sockets[user];
                        delete socketToFB[user];
                        delete guesses[user];
                        delete partners[user];
                        delete imageIDsSeen[user];
                        noOfUsers--;
                        console.log('Number of users: ' + noOfUsers);
                        socket.broadcast.emit('users', currentUsers);
                }
        });

   socket.on('closing', function(data) {
                var i = currentUsers.indexOf(data);
                if (i != -1) { // So it won't do this if the user hasn't started a game
                        var partner = partners[data];
```

```
                        sockets[partner].emit('partner quit');
                        currentUsers.splice(i, 1);
                        imageIDs.splice(i, 1);
                        i = currentUsers.indexOf(partner);
                        currentUsers.splice(i, 1);
                        imageIDs.splice(i, 1);
                        delete sockets[data];
                        delete socketToFB[data];
                        delete guesses[data];
                        delete partners[data];
                        delete imageIDsSeen[data];
                        delete sockets[partner];
                        delete socketToFB[partner];
                        delete guesses[partner];
                        delete partners[partner];
                        delete imageIDsSeen[partner];
                        noOfUsers -= 2;
                        console.log('Number of users: ' + noOfUsers);
                        socket.broadcast.emit('users', currentUsers);
                }
    });
});
```

index.ejs

```
<!DOCTYPE html>
<html xmlns:fb="http://ogp.me/ns/fb#" lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-
scale=2.0, user-scalable=yes" />

    <title><%= app.name %></title>
    <link rel="stylesheet" href="stylesheets/screen.css" media="Screen" type="text/css" />
    <link rel="stylesheet" href="stylesheets/mobile.css" media="handheld, only screen and
(max-width: 480px), only screen and (max-device-width: 480px)" type="text/css" />

    <!--[if IEMobile]>
    <link rel="stylesheet" href="mobile.css" media="screen" type="text/css"  />
    <![endif]-->

    <!-- These are Open Graph tags.  They add meta data to your  -->
    <!-- site that facebook uses when your content is shared     -->
    <!-- over facebook.  You should fill these tags in with      -->
    <!-- your data.  To learn more about Open Graph, visit       -->
    <!-- 'https://developers.facebook.com/docs/opengraph/'       -->
    <meta property="og:title" content="<%= app.name %>" />
    <meta property="og:type" content="website" />
    <meta property="og:url" content="<%= url() %>" />
    <meta property="og:image" content="<%= url('/logo.png') %>" />
    <meta property="og:site_name" content="<%= app.name %>" />
    <meta property="og:description" content="My first app" />
    <meta property="fb:app_id" content="<%= app.id %>" />

    <script type="text/javascript" src="/scripts/jquery.min.js"></script>

    <script src="/socket.io/socket.io.js"></script>

    <script>
    var socket = io.connect();
    var fbid;
    var partner;
    var points = 0;

    var count = 90;

    function startAGame() {
       FB.api('/me', function(response) {
             fbid = response.id;
             socket.emit('name', fbid);
       });
       document.getElementById("waiting").innerHTML = "Please wait...";
       document.getElementById("startAGameButton").style.visibility = "hidden";
```

```
    socket.on('done it', function() {
        document.getElementById("waiting").style.visibility = "hidden";
        document.getElementById("pass").style.visibility = "visible";
        document.getElementById("time").style.visibility = "visible";
        document.getElementById("points").style.visibility = "visible";
        document.getElementById("tag").style.visibility = "visible";
        document.getElementById("submitTheTag").style.visibility = "visible";
        document.getElementById("tagText").style.visibility = "visible";
    });
}

var counter;
socket.on('start timer', function() {
    counter = setInterval(timer, 1000);
});

function timer() {
    count -= 1;
    if (count == 0) {
        socket.emit('time up', points);
        clearInterval(counter);
    }
    document.getElementById("time").innerHTML = "Seconds left: " + count;
}

socket.on('r', function(data) {
    var url = "http://res.cloudinary.com/hd93ufv7u/image/upload/" + data + ".jpg";
    document.getElementById("image").src=url;
    document.getElementById("tagText").innerHTML = "Tags entered:<br />";
    document.getElementById("imagesShown").innerHTML += data + " ";
    document.getElementById("passed").innerHTML = "";
});

socket.on('taboo words', function(data) {
    var string = "Taboo:<br />";
    for (i = 0; i < data.rows.length; i++) {
        string = string + data.rows[i].tag_text;
        string = string + "<br />";
    }
    document.getElementById("taboo").innerHTML = string;
});

socket.on('is a taboo', function() {
    document.getElementById("passed").innerHTML = "That word is a taboo word!";
});

socket.on('creator', function(data) {
    document.getElementById("creator").innerHTML = "Image courtesy of
FreeDigitalPhotos.net/" + data;
});
```

```
    socket.on('passed', function() {
        document.getElementById("passed").innerHTML = "Your partner has passed on this
image!";
        document.getElementById("confirmPass").style.visibility = "visible";
    });

    function passOnThisImage() {
        socket.emit('pass');
    }

    function confirmPassOnThisImage() {
        document.getElementById("confirmPass").style.visibility = "hidden";
        document.getElementById("passed").innerHTML = "";
        socket.emit('confirm');
    }

    function submitTag() {
            document.getElementById("passed").innerHTML = "";
        var tag = document.getElementById("tag").value;
        document.getElementById("tagText").innerHTML += tag;
        document.getElementById("tagText").innerHTML += "<br />";
        document.getElementById("tag").value = "";
        socket.emit('tag', tag);
    }

    socket.on('users', function(data) {
        var string = "";
        for (i = 0; i < data.length; i++) {
            string = string + data[i];
            string = string + " ";
        }
        document.getElementById("users").innerHTML = string;
    });

    socket.on('partner', function(data) {
        var partner = data;
        document.getElementById("conn").innerHTML = "You are now connected to " +
partner;
    });

    socket.on('already here', function() {
        alert("You've already logged into the game!");
    });

    socket.on('matched', function(data) {
        var string = "Points: ";
        points += 100;
        string = string + points;
        document.getElementById("points").innerHTML = string;
```

```
    document.getElementById("imagesShown").innerHTML += data + " ";
  });

  socket.on('redirect', function() {
    document.getElementById("time").style.visibility = "hidden";
    document.getElementById("points").style.visibility = "hidden";
    document.getElementById("creator").style.visibility = "hidden";
    document.getElementById("taboo").style.visibility = "hidden";
    document.getElementById("image").style.visibility = "hidden";
    document.getElementById("tagText").style.visibility = "hidden";
    document.getElementById("tag").style.visibility = "hidden";
    document.getElementById("submitTheTag").style.visibility = "hidden";
    document.getElementById("pass").style.visibility = "hidden";
    document.getElementById("passed").style.visibility = "hidden";
    document.getElementById("confirmPass").style.visibility = "hidden";

    document.getElementById("finishScreen").innerHTML = "Time's up!<br /><br
/>Points: " + points;
  });

  function closing() {
    socket.emit('closing', fbid);
  }

  socket.on('partner quit', function() {
        document.getElementById("time").style.visibility = "hidden";
        document.getElementById("points").style.visibility = "hidden";
        document.getElementById("creator").style.visibility = "hidden";
        document.getElementById("taboo").style.visibility = "hidden";
        document.getElementById("image").style.visibility = "hidden";
        document.getElementById("tagText").style.visibility = "hidden";
        document.getElementById("tag").style.visibility = "hidden";
        document.getElementById("submitTheTag").style.visibility = "hidden";
        document.getElementById("pass").style.visibility = "hidden";
        document.getElementById("passed").style.visibility = "hidden";
        document.getElementById("confirmPass").style.visibility = "hidden";

    document.getElementById("finishScreen").innerHTML = "Sorry, your partner quit";
    document.getElementById("finishScreen").style.left = "150px";
    clearInterval(counter);
  });

  window.onbeforeunload = closing;

  </script>

  <script type="text/javascript">
    function logResponse(response) {
      if (console && console.log) {
```

```
    console.log('The response was', response);
  }
}

$(function(){
  // Set up so we handle click on the buttons
  $('#postToWall').click(function() {
    FB.ui(
      {
        method : 'feed',
        link   : window.location.href
      },
      function (response) {
        // If response is null the user canceled the dialog
        if (response != null) {
          logResponse(response);
        }
      }
    );
  });

  $('#sendToFriends').click(function() {
    FB.ui(
      {
        method : 'send',
        link   : window.location.href
      },
      function (response) {
        // If response is null the user canceled the dialog
        if (response != null) {
          logResponse(response);
        }
      }
    );
  });

  $('#sendRequest').click(function() {
    FB.ui(
      {
        method  : 'apprequests',
        message : $(this).attr('data-message')
      },
      function (response) {
        // If response is null the user canceled the dialog
        if (response != null) {
          logResponse(response);
        }
      }
    );
  });
```

```
    });
  </script>

  <!--[if IE]>
    <script type="text/javascript">
      var tags = ['header', 'section'];
      while(tags.length)
        document.createElement(tags.pop());
    </script>
  <![endif]-->
</head>
<body>
  <div id="fb-root"></div>
  <script type="text/javascript">
    window.fbAsyncInit = function() {
      FB.init({
        appId      : '<%= app.id %>', // App ID
        channelUrl : '<%= url_no_scheme('/channel.html') %>', // Channel File
        status     : true, // check login status
        cookie     : true, // enable cookies to allow the server to access the session
        xfbml      : true // parse XFBML
      });

      // Listen to the auth.login which will be called when the user logs in
      // using the Login button
      FB.Event.subscribe('auth.login', function(response) {
        // We want to reload the page now so PHP can read the cookie that the
        // Javascript SDK sat. But we don't want to use
        // window.location.reload() because if this is in a canvas there was a
        // post made to this page and a reload will trigger a message to the
        // user asking if they want to send data again.
        window.location = window.location;
      });

      FB.Canvas.setAutoGrow();

      document.getElementById("startAGameButton").style.visibility = "visible";
    };

    // Load the SDK Asynchronously
    (function(d, s, id) {
      var js, fjs = d.getElementsByTagName(s)[0];
      if (d.getElementById(id)) return;
      js = d.createElement(s); js.id = id;
      js.src = "//connect.facebook.net/en_US/all.js";
      fjs.parentNode.insertBefore(js, fjs);
    }(document, 'script', 'facebook-jssdk'));
  </script>

  <header class="clearfix">
```

```
    <% if (user) { %>
    <p id="picture" style="background-image: url(https://graph.facebook.com/<%= user.id
%>/picture?type=normal)"></p>

    <div>
      <h1>Welcome, <strong><%= user.name %></strong></h1>
      <p class="tagline">
        This is your app
        <a href="<%= app.link %>" target="_top"><%= app.name %></a>
      </p>

      <div id="share-app">
        <p>Share your app:</p>
        <ul>
          <li>
            <a href="#" class="facebook-button" id="postToWall" data-url="<%= url() %>">
              <span class="plus">Post to Wall</span>
            </a>
          </li>
          <li>
            <a href="#" class="facebook-button speech-bubble" id="sendToFriends" data-
url="<%= url() %>">
              <span class="speech-bubble">Send Message</span>
            </a>
          </li>
          <li>
            <a href="#" class="facebook-button apprequests" id="sendRequest" data-
message="Test this awesome app">
              <span class="apprequests">Send Requests</span>
            </a>
          </li>
        </ul>
      </div>
    </div>
    <% } else { %>
    <div>
      <h1>Welcome</h1>
      <div class="fb-login-button" data-scope="user_likes,user_photos"></div>
    </div>
    <% } %>
  </header>

  <section id="guides" class="clearfix">
  <p id="finishScreen"></p>

  <p id="time" style="visibility:hidden">Seconds left: 90</p>
  <p id="points" style="visibility:hidden">Points: 0</p>
  <p id="creator"></p>
  <p id="taboo"></p>
  <img id="image">
```

```
    <p id="tagText" style="visibility:hidden">Tags entered:<br /></p>
    <button type="button" id="startAGameButton" onclick="startAGame()"
style="visibility:hidden"></button>
    <p id="waiting"></p>
    <div id="formDiv">
    <form>
        <input type="text" id="tag" style="visibility:hidden"><br />
    </form>
    <button type="button" id="submitTheTag" onclick="submitTag()"
style="visibility:hidden"></button>
    <button type="button" id="pass" onclick="passOnThisImage()"
style="visibility:hidden"></button>
    </div>
    <div id="passDiv">
    <p id="passed"></p>
    <button type="button" id="confirmPass" onclick="confirmPassOnThisImage()"
style="visibility:hidden"></button>
    </div>
    <div id="debugDiv">
    <h6>List of users:</h6>
    <p id="users"></p>
    <p id="conn"></p>
    <p id="imagesShown"></p>
    </div>
    </section>

  </body>
</html>
```

Appendix B

Here we shall detail an example of how we tested the system for correct tag processing. First, we recorded the contents of the various database tables (usually in the form of an Excel workbook). Then, we played a game and recorded what should have happened during that game:

20 blue 18 2 7 union jack 16 money 17 maths 30 star 5 inside 10

This is the output we obtained during the game. Eliminating the times where we 'passed' on an image or when we didn't have time to enter a tag at the end we get:

20 blue 7 union jack 16 money 17 maths 30 star 5 inside

Then, we look up the IDs for each tag (the numbers above are the image IDs), and we get:

20	63
7	75
16	56
17	66
30	48
5	73

Now, looking at the data we gathered before, we can work out what should have happened:

Changes made to *non_taboo* (in the format image_id, tag_id, agrees_left):
20, 63, 5 → 20, 63, 4
17, 66, 3 → 17, 66, 2
30, 48, 3 → 30, 48, 2
5, 73, 5 → 5, 73, 4

Additions to *non_taboo*:
7, 75, 5

Deletions to *non_taboo* (and additions to *taboo*):
16, 56, 1 (inserted into *taboo* as 16, 56 (image_id, tag_id))

We check that these changes have happened to the tables, and if so, then the tags were processed as expected.

References

1. von Ahn, L and Dabbish, L. 2004. Labeling Images with a Computer Game. *ACM Conference on Human Factors in Computing Systems, CHI 2004.* Vienna, Austria, 24-29 April, 2004. pp 319 – 326.

2. GWAP. *gwap.com - ESP Game.* Available at: http://www.gwap.com/gwap/gamesPreview/espgame/ [Accessed: 27th April 2013].

3. Joyent. *node.js*. Available at: http://nodejs.org/ [Accessed: 5th December 2012].

4. Rauch, G. *Introducing Socket IO.* Available at: http://socket.io/ [Accessed: 7th December 2012].

5. Heroku. *Heroku*. Available at: http://www.heroku.com/ [Accessed: 7th December 2012].

6. Cloudinary. *Cloudinary - Image Management in the Cloud*. Available at: http://cloudinary.com/ [Accessed: 27th April 2013].

7. Heroku. *Heroku Postgres*. Available at: https://devcenter.heroku.com/articles/heroku-postgresql [Accessed: 7th December 2012].

8. FreeDigitalPhotos.net. *Use of free images - how to acknowledge the image creator.* Available at: http://www.freedigitalphotos.net/images/acknowledgement.php [Accessed: 27th April 2013].

9. Facebook. *JavaScript SDK - Facebook developers*. Available at: https://developers.facebook.com/docs/reference/javascript/ [Accessed: 27th April 2013].

10. White, T. 2004. *Can't beat Jazzy.* Available at: http://www.ibm.com/developerworks/java/library/j-jazzy/ [Accessed: 1st May 2013].

11. Freelon, D. *ReCal3: Reliability for 3+ Coders*. Available at: http://dfreelon.org/utils/recalfront/recal3/ [Accessed: 13th December 2012].

12. pgAdmin. *pgAdmin*. Available at: http://www.pgadmin.org/ [Accessed: 7th December 2012].