Advanced Web Technologies - Say The Same Thing

Zsolt Varga
40212393@napier.ac.uk
Edinburgh Napier University - Advanced Web Technologies (SET09103)

Abstract

Say The Same Thing is a word game and originally smartphone app developed by OK Go. In this project, we attempted to recreate this game as a web app for the browser, using standard technologies on the front end (HTML, javascript, CSS), and the python Flask microframework on the server side, with the help of a few additional libraries. We created a fully functional online game, with its own API, and a pleasant user interface.

Keywords – coursework, Edinburgh, napier, university, zsolt, varga, web, technologies, front-end, back-end, server, flask, js, javascript, html, css, Say The Same Thing, app, website, dynamic

1 Introduction

Say the same thing is an interactive web app and browser game that uses the python Flask micro framework. It was created as a coursework assignment for the Edinburgh Napier University. The idea and the basic functionality of this game is based on the mobile app and game Say the same thing. This game was originally developed by OK Go (Figure 18) and we do not claim any copyrights - the project is being used for educational purposes only.

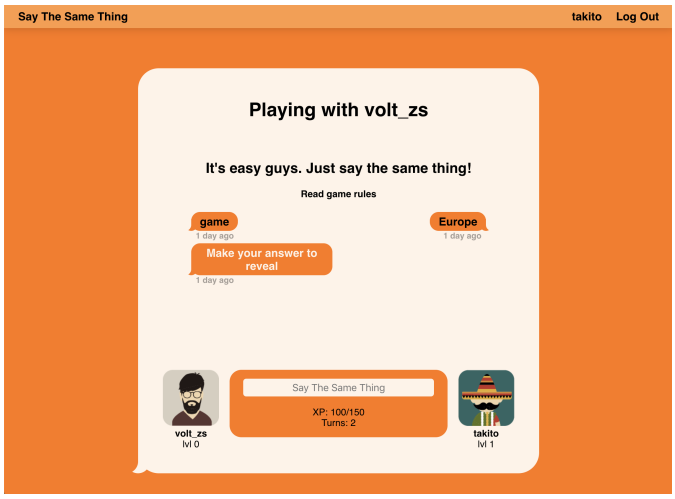

Figure 1: The page of a game of Say The Same Thing between two players

The basic idea of the game, as the name suggests, is to say the same thing. There are exactly two players in each game and these players take turns. In each turn, the user says a word in the hopes of their words being the same. In the first turn, the choice of words is completely random, however, in each subsequent turn the players aim to say the word that "connects" the two words from the previous round. (Figure 1) The game goes on until the same word is said at the same time.

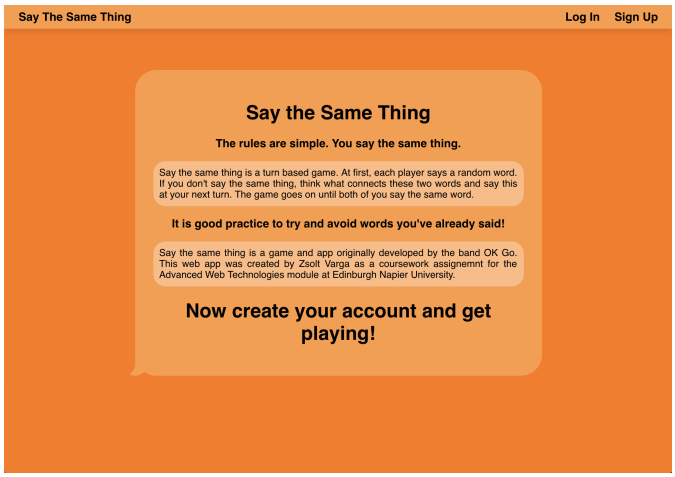

Figure 2: The welcome page as viewed when not logged in

Besides the core game mechanics, the web app also allows basic user functionality, and allows users to level up by gaining XP points in games, add buddies to easily "save" players, and manage their list of games.

The technologies used for the web app's server side are quite straightforward - python Flask using SQLite as the database. However, several libraries have been installed, with some being more significant than others.
For handling the database and creating an abstraction level on top of it, the flask-SQLAlchemy library was chosen. This was done to avoid writing too much pure SQL as the focus of this project was not demonstrating knowledge of SQL, and also saved some time in general. To manage users and make user interactions secure, the flask-login library does the job rather well. And lastly, the security module of the werkzeug utility library was used for password hashing.
Besides the basic technologies on the front-end, the web app also makes use of the Jinja templating engine and some of jQuery's ajax calls.

2 Design

2.1 User Interface

For the design of the UI and the overall layout of the web app, several factors had to be examined. It was our aim to create a pleasant looking website with a nice user experience. Most of the design elements of the app are based on the "word game" nature of Say the same thing. The most prominent buttons and containers appear as speech bubbles with rounded corners, therefore, we avoided using any rough edges on any elements to add to the illusion of a "chat". The speech bubbles also have a practical use within the game itself (see figure 3, where they indicate exact turns and time.

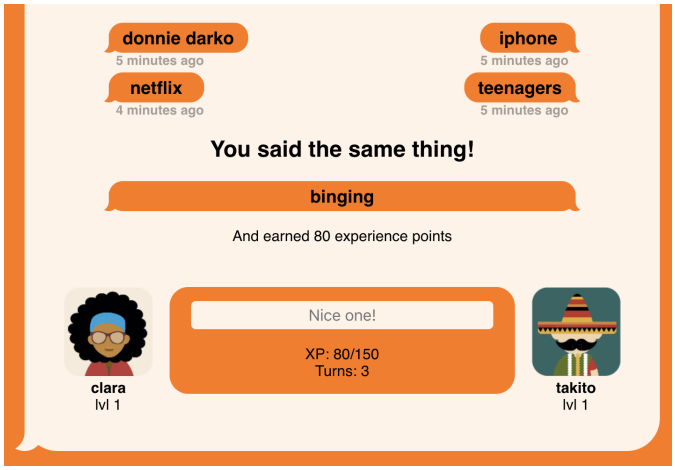

Figure 3: A closer look at the speech bubbles in a won game

The user functionality has been considered before designing the layout of the app. As a result, the top navigation bar gives easy access to the user's notifications, a link to the home page and a logout function. When inside the game, the navigation bar is the only other element besides the play window to emphasise and differentiate this page, however, for every other page the user has their own sidebar providing easy access to core functionality (see figure4).

As the sidebar acts as the user's personal navigation tool, all the functionality is directly tied to their account - the level bar and avatar indicate their progress and link to their user settings, a link to their buddies, their games (figure 13), and a play button that starts a new game with a random player. The left hand side of the page is the main container for the content and changes accordingly (for examples see figures 5, 15, and 14).
Generally, the layout for each page is very similar and mostly fixed, however, there are a couple of templates that can be viewed when both signed in and not. In these cases the user sidebar is displayed accordingly (compare figures 2 and 11).

To honour the original app, a similar orange tone has been used as the main colour. Accompanied by the mostly flat design with minimal shading and simple animation, the website has a very 80's feel which works well with being a word game. Despite the flat design, some of the elements have a drop shadow to indicate they are "floating" above the content. (figure 6) The flat icon user avatars also add to the experience - these were obtained from webdesignerdepot.com [1] as a free to use resource.

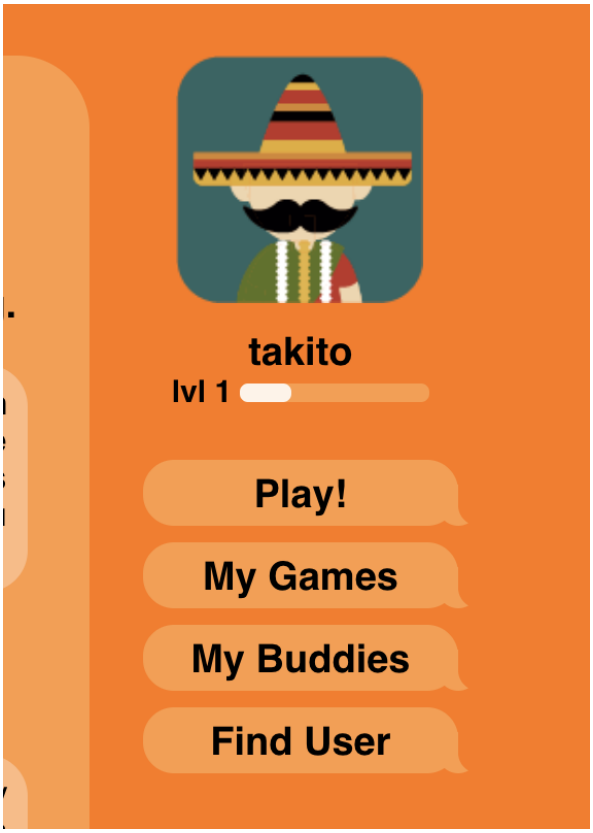

Figure 4: The user's sidebar that appears on most of the pages when logged in

Figure 6: The floating notification window of a user

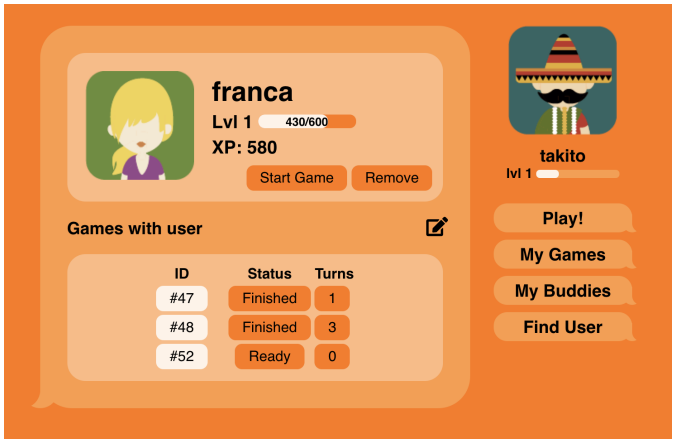

Figure 5: Page for viewing a specific user

2.2 Routing and Authentication

Coming up with the routing hierarchy was quite a challenging task, considering the amount of functionality and different ways to interact with the flask app.

Users are able to perform all CRUD operations on their own accounts, add and remove buddies, create and delete games. For these routes, putting them in a hierarchy was quite straightforward.

There are several views that are "atomic" and therefore couldn't really be placed in a branch of the routing tree. These include the login/logout/register views, but also routes such as /clear_notifications, or /play_random - where the user waits for a connection to a random player.

Some of the routes only provide data in the form of strings or JSON. These routes were used for polling the database for game or notification data. To see all routes and their description see figure 7.

ROUTE	NOTE
* /	welcome page
* /register	register a new user
* /login	sign in existing user
/logout	log out existing user
/buddies	all buddies of current user
/games	all games of current user
/game/<game_id>	specific game - this is the play route
/game/<game_id>/remove	delete specific game
/game/<game_id>/submit_answer	submit answer to a specific game
/play_random	(wait to) connect to a random player
/start_game/<username>	start a new game with specific user
/users	show recent users and search for users
/user/<username>	show specific user
/user/<username>/add_buddy	add specific user as a buddy
/user/<username>/remove_buddy	remove specific user from buddies
/user_settings	settings/user page of the current user
/user_settings/change_password	change password of current_user
/user_settings/change_avatar	change avatar of current user
/user_settings/delete_account	delete the current user
/clear_notifications	mark all notifications of current user as viewed
/poll_game	open request to wait for opponent response
/find_waiting_user	open request to wait for a player
/check_notifications	check if the number of new notifications has changed
/get_notifications	retrieve a list of notification objects

Figure 7: List of all routes and their description. *indicates routes that do not have a @login_required decorator

As many of the routes provide a way for the user to modify data in the database, authorisation and authentication played a big role during development. Thanks to the flask-login library, almost every route is wrapped in a @login_required decorator, which prevents anyone who's not logged in to access data about users. Subsequently, for every view that handles data in a way, it was necessary to check whether the data belongs to the current user.

For instance, within the view for removing a game (figure 8), it is first checked whether the request is coming from a logged in user, then if the game to be deleted is found, it is compared against the current user's games before being deleted, otherwise a 403 Forbidden response is sent.

```
@app.route('/game/<game_id>/remove')
@login_required
def delete_game(game_id):
    game = Game.query.get(game_id)
    if not game:
        return abort(404, "Game not found"), 404
    if not game in current_user.games:
        abort(403, "You are not authorised to delete this game."), 403
    db.session.delete(game)
    db.session.commit()
    next = request.args.get('next')
    session['deletemode'] = True
    if next:
        return redirect(next)
    return redirect(url_for('my_games'))
```

Figure 8: The floating notification window of a user

2.3 Error Handling

A single custom template is used to handle all the implemented errors. See figure 17. This is done by all errorhandlers referencing the same function, which returns a rendered template with the passed in error message. This way, whenever a player bumps into an error, the user experience is not compromised despite the interruption.

3 Enhancements

Regardless of the web app and game being fully functional, there could be several enhancements that could transform the app to a whole new experience.

3.1 Game Modes

One way of doing this would be introducing different game modes. Currently, the user are able to type in any word of their choice. Besides disallowing the players from using previously mentioned words, they could also choose between different modes, for instance a kids' game where they choose from a pool of words or images. Alternatively, a vocabulary game could be introduced where players compete against each other and try to be the first to guess a niche word based on clues.

3.2 Error handling

Currently, the web app only handles three different error codes (403, 404, and 405). Considering the nature of the web app and the functionality, these are the three most relevant ones, however, this could be largely expanded, and more types of responses should be provided.

3.3 User Interaction

Users already have several channels for interaction, however there is no direct way of communicating between users. Surely, a full-fledged chat is unnecessary, however, it would be useful if users could send each other stickers from a selection of emotions/questions such as "One more game?", for example. Similarly, as the whole game is visually reminiscent of an online conversation, it would be a nice touch to be able to react to an opponent's answer in a game with an emoji - just like in Facebook's Messenger.

3.4 Trophies

Adding the ability to gain trophies from game achievements might also increase the players' engagement in the app, since not everyone might be interested in the currently implemented leveling scheme. Gaining rewards for "one turn wins" or "not saying the same word twice" could be examples of these trophies.

3.5 Code separation

A slightly less conceptual enhancement could be a better separation of code and concerns. There are already a few measures in place (more in Critical Evaluation), however, separating code that concerns the game logic into a class of its own might be beneficial and clean some of the mess up.

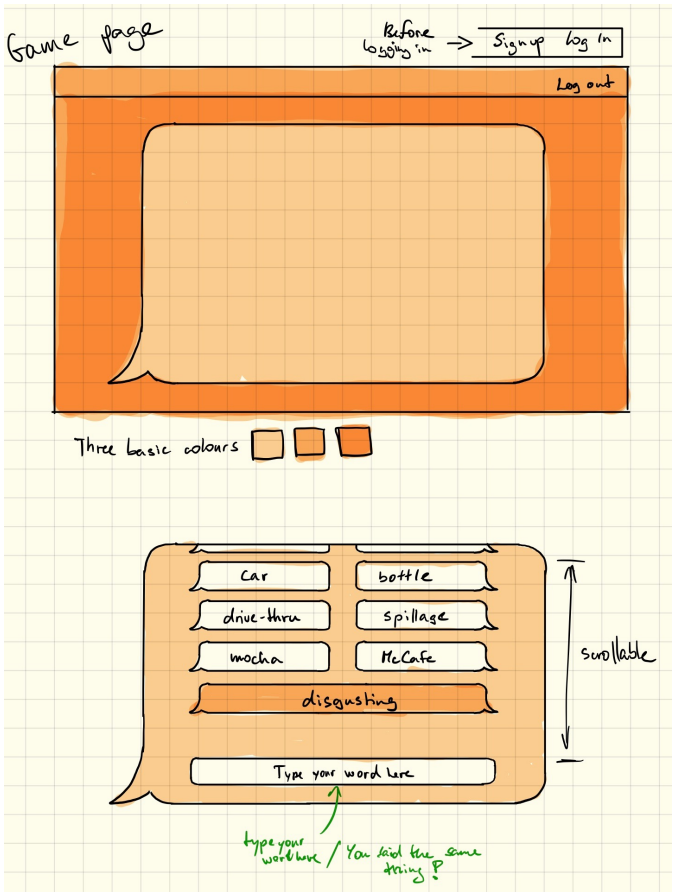

Figure 9: Initial sketch of the UI

4 Critical Evaluation

In overall, the web app has been fully implemented as per initial design (see figures 9, 12) and has all the working functionality as per plan. Regardless, there is always space for improvement, which would reflect in a longer time for development. Here we will discuss some of the most crucial features and some of the problems with their implementation.

4.1 Project structure

One of the first considerations when developing was the structure of the project. As there are not that many types of views in this small app, these were kept together in the stst_app.py file. However, every other files are kept in a separate stst_project folder, including the __init__.py file. This way, the stst_project is recognised as a library by python and is straightforward to import objects/classes into our flask app (see figure 10).
For the next iteration of the project, as more and more views are adding up, it would be sensible to separate these into their own files. There are, however, a great number of functions within our stst_app.py file that do not act as views, and only provide logic for processing user and game data. These should very easily be put in their own class so as to not clutter the flask app.

```
from stst_project import app, db
from stst_project.models import User, Game, Turn, Notification
```

Figure 10: Importing the database Models and other objects from the stst_project folder - treated as a library

4.2 Database Implementation

Since our web app has rather extensive user support, it was necessary to store user data - this was done via an SQLite database using the flask-SQLAlchemy as an abstraction layer. We have made an effort to avoid storing any data that is unnecessary, mostly for security purposes but also as a way to implement a relational database the right way. For example, the user's level is not stored in the database, as this is easily calculated based on their XP points and would be redundant information.
Thanks to the SQLAlchemy library, creating Models/relations and then retrieving data by using these was straightforward and quick, as opposed to having to write SQL statements for every query. The models have been placed in a separate models.py file inside the project folder for better separation of concerns. This implementation works well and is also one of the cleanest solutions in the app code-wise.

4.3 Polling and Long Polling

Naturally, the main focus of the app is the game functionality. Users have the option to start a game with any random online player, or a specific user/buddy. When playing with a random user, the server needs to check what other users are currently waiting to start a game as well, and respond based on that information. This requires a real time two-way

interaction between the client and server which is not truly accomplishable via the HTTP protocol.
One way of solving this problem could be using websockets and therefore change the protocol itself or use long-polling. As a less time-consuming solution, we chose to implement long polling, which as opposed to polling doesn't clutter the server with requests as much but does require multi-threading on the server side. This way, the user sends a request to the server, which in return keeps the request open and runs a loop that stops and returns a response whenever a change occurs in the database - in this case when another user indicates they are waiting for a game.

A similar solution was used to pull notifications from the server. However, as it is less important for the notifications to be real-time, the client side javascript sends a simple request to the server to check for notifications every 20 or so seconds.

By utilising polling and long polling, there is a certain trade-off and a balance that needs to be achieved. However, if the application was used by a larger userbase, it would be a better solution to use a websocket.

4.4 Cookies/session

The flask app makes use of the session object to a certain extent. For instance, when registering, the session saves the details of the form (only the ones that do not contain sensitive information eg. passwords), so that the user does not need to fill everything out in case they get redirected back due to an error (for example a taken username).
However, this could be utilised for many more purposes, such as storing the last few url's so that some redirects may not be hard coded.

5 Personal Evaluation

Generally speaking, working on this project was a fun and new experience. Flask is a very intuitive framework and provides great functionality for creating an API. When diving into the project, python definitely was not one of my favourite languages, however, it seems to have its own charm and despite the simple syntax can be quite challenging and fun.

At the time of deciding about the nature of this project, it was definitely not clear how all the functionality will be achieved, but I am pleasantly surprised at the solid piece of software that has come out of this project. All of the initially planned functionality was fully developed, and several new technologies were learned on the way.
For example, figuring out how the flask-SQLAlchemy library worked was a small challenge of its own, especially the many-to-many and the self-referencing relationships the User model required.

Planning is always crucial, but at the same time a lot of obstacles have forced a new plan. For instance, the routing hierarchy has been modified several times whenever a new view that didn't quite fit in was added. Same was the case with structure and inheritance of the templates.

Besides the material and advice provided as part of the module, a lot of background learning had to be done to fully understand the technologies used, especially the libraries not covered by the modules. A great deal of time was spent by completing the Udemy course Python and Flask Bootcamp: Create Websites Using Flask! [2]

Developing this project took hours of planning, coding, and figuring out why things don't work the way they should. It was a good and rewarding experience, and despite there being a large room for improvement, the project was definitely a successful attempt at creating a fully functional and fun online word game.

References

[1] W. D. Depot, "Free download: Avatar vector collection." https://www.webdesignerdepot.com/2016/06/free-download-avatar-vector-collection/. Accessed: 2018-07-11.

[2] J. Portilla, "Python and flask bootcamp: Create websites using flask." https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/v4/overview. Accessed: 2018-09.

6 Appendices

6.1 Library Installation

The requirements to run this flask app (besides the Flask framework) include a few addition libraries. These should be installed via pip and are the following:

- flask_login
- Werkzeug
- Flask-SQLAlchemy

6.2 Screenshots and Images

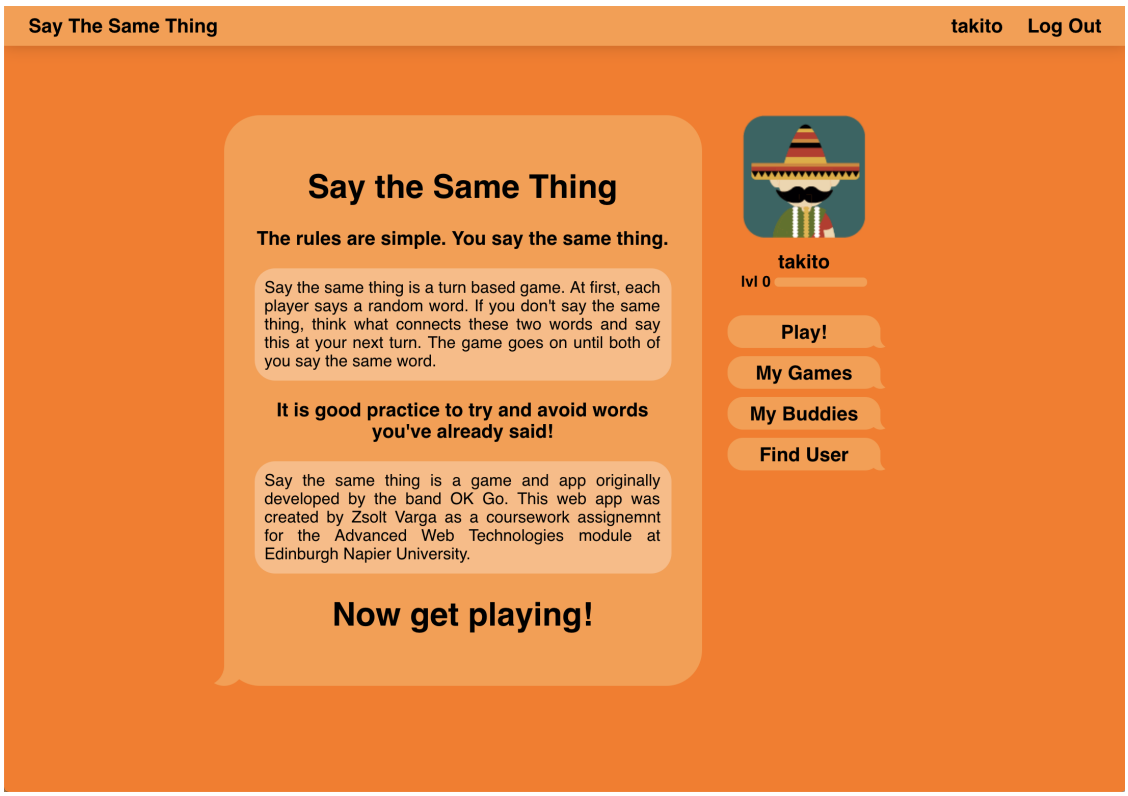

Figure 11: The welcome page as viewed by a logged in user

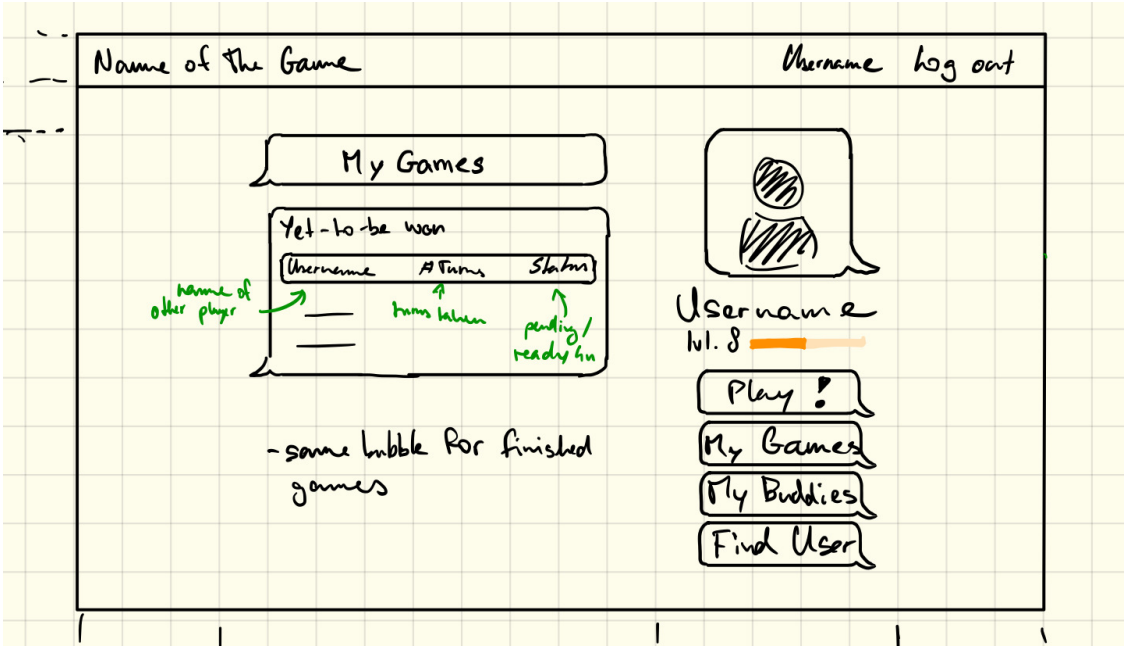

Figure 12: Initial sketch of the app layout

hh

Figure 13: Rendered template of the user's games page

Figure 14: Page for recent users and user search

Figure 15: Viewing the logged in user's own page and account options

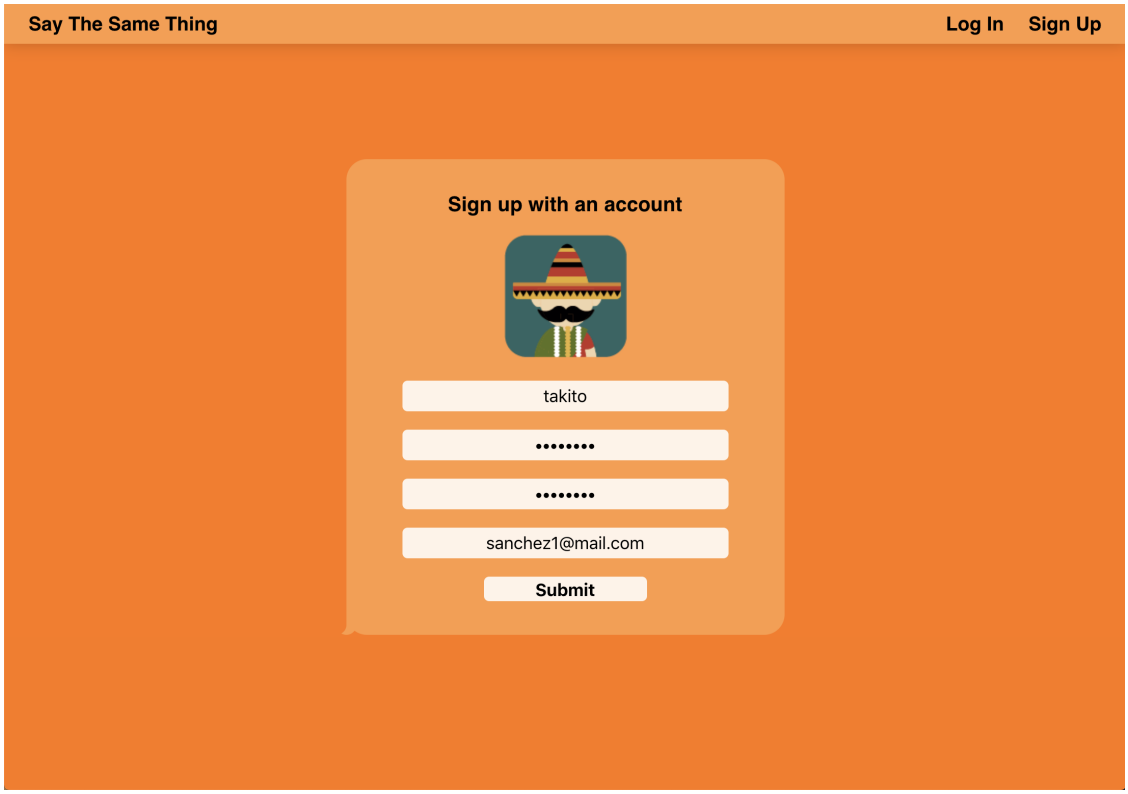

Figure 16: The register page

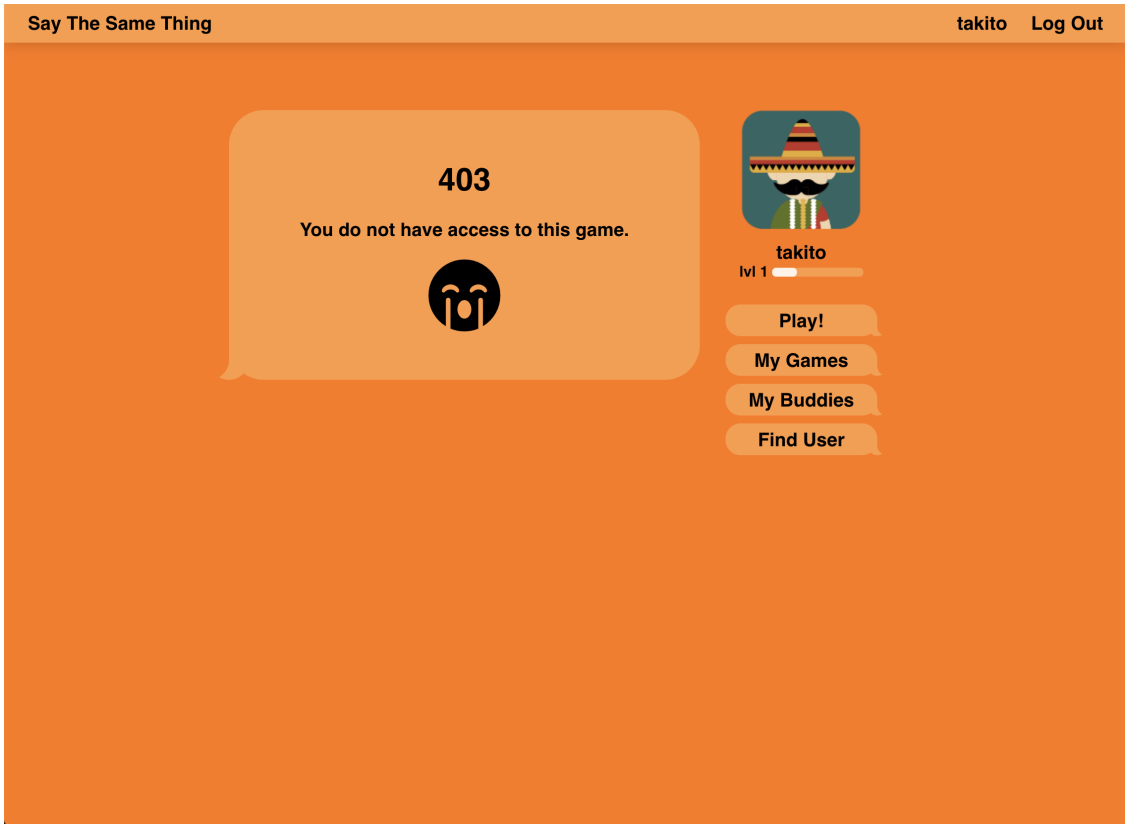

Figure 17: Rendered error page template with the code and message passed in

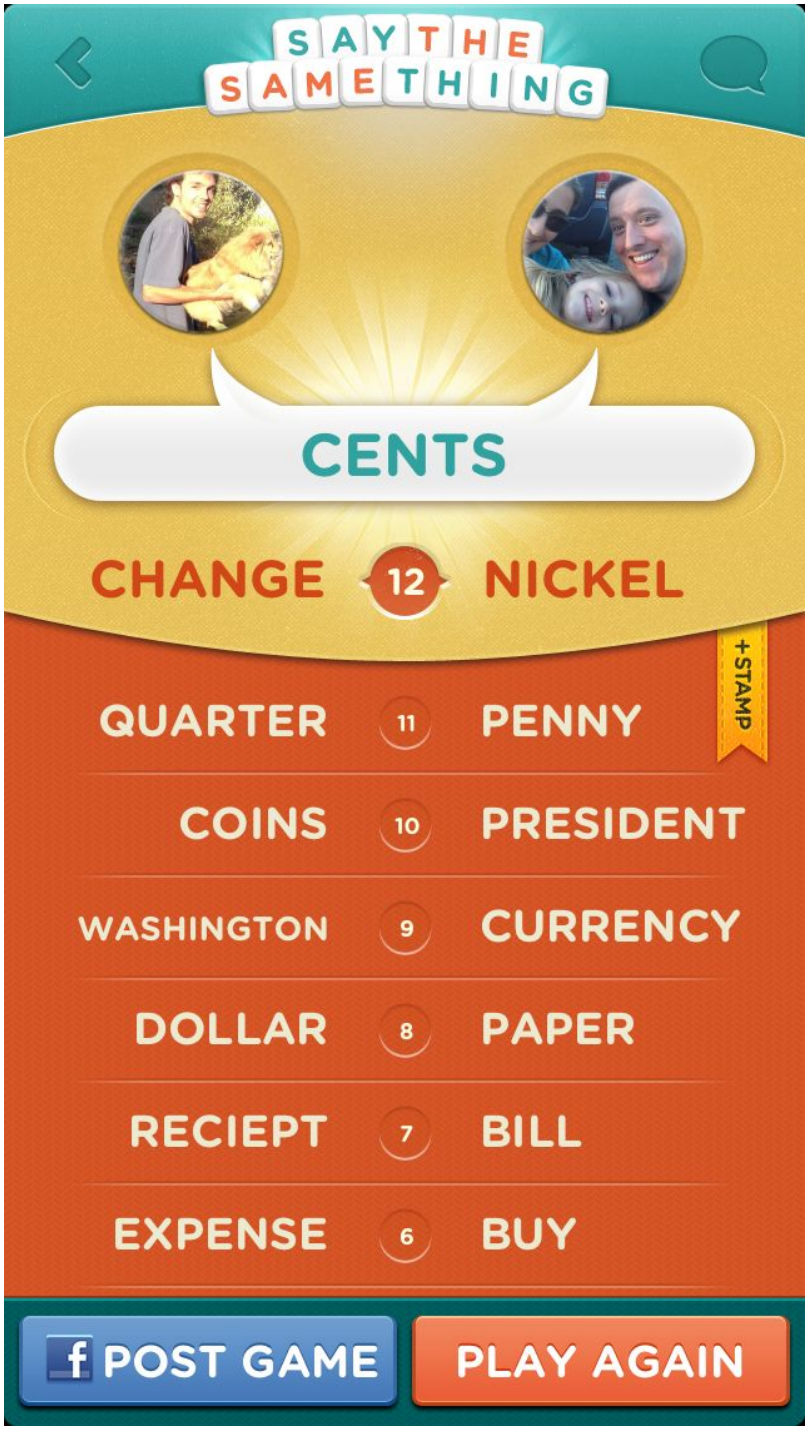

Figure 18: The Say the Same Thing app originally developed by OK Go