Smashing Word Constructor
Design and Planning Document
3/4/2013
Version 1.0 - Initial Version

Table Of Contents

Document Revision History

Revision 1.0 2013-03-04 - initial version

System Architecture

Overview

'Smashing Word Constructor' application is to run natively on an iOS device. It will be using the 3-tier architecture which consists of a front-end iOS platform, a backend Application server and a Database tier that are maintained as independent modules. Therefore, data flow will only occur between neighboring layers to promote modularity. The responsibility breakdown for each of the layers is as follows:

- iOS platform - responsible for all of the actual user interface/interaction and will consists of all the user's View and connects to the Application server using HTTP Protocol when such service is needed
- Application server - responsible for providing constant connections with multiple clients (iOS devices) which contains the business logic of the app as well as acting as a middleware between the clients and the database.
- Database - responsible for storing and providing access to the application data

MTV Architecture

The Application server layer will be using MTV(or MVC[1]) design pattern - Model, Template, View. The responsibility breakdown for each of the components is as follows:

- Model - responsible for the database models, business logic and functions and communicating with the database for any necessary actions
- Template - will not be implemented
- View - responsible for handling data or events between front-end and application server and converting it to commands for the Model

[1] MVC, http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Details of MTV components

Model

The Model component will be the only source of communication with the database in terms of database schema, manipulating data in the database and providing service for the business logic requested by the View component. It will implement functions for testing data input correctness as well as performing queries and retrieving data from the database for the View component. The Model component will act as the only middleware between View component and the database.

Template

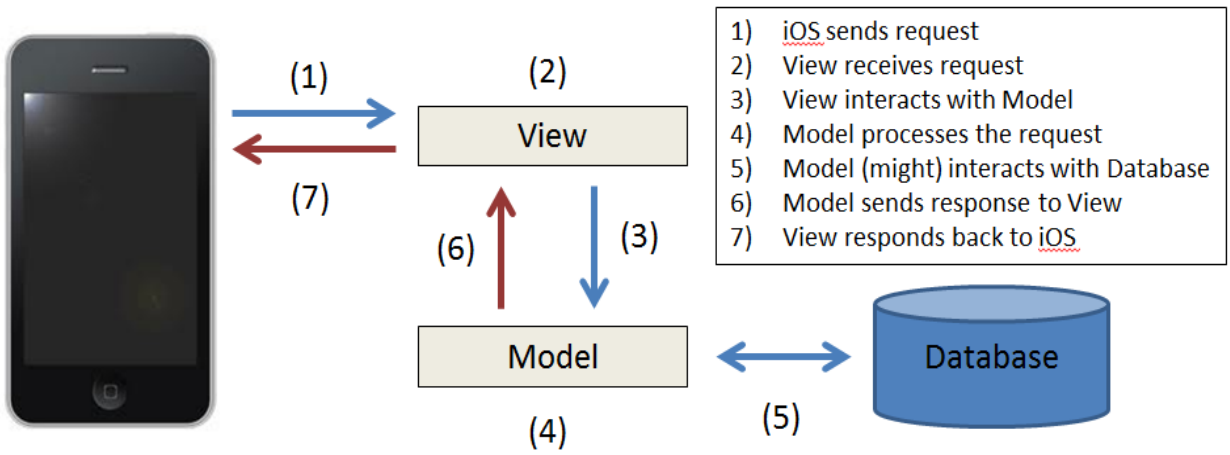
The Template component will not be implemented as this app will solely be run on an iOS device.

View

Communication between front-end and application will go through the View component. The View component will open up a TCP connection through a HTTP protocol and accept client's requests in JSON which will then be parsed. Then, it will generally follow these steps:

1. View passes the data to Model
2. Model will process the data by calling related functions and performing business logic
3. Model will return a JSON data to View
4. View will send the JSON request/data back to the client

Diagram of MTV Architecture Overview



Design Details

Frontend iOS Details

This app will be using the NextStep (NS) JSON and HTTP libraries in Xcode to implement all the Views (pages) and function logic. Client-Server communication will solely be using the JSON data via HTTP.

In addition to these, an offline dictionary of English words will be needed. (The team has agreed upon the use of Lexicontext which provides more than 150,000 words and great features such as fast data access and small memory usage.) (assuming we're using this)

Application Server Details

The server will be built in Django (a Python web framework designed using the Model-Template-View pattern). It will be deployed on Heroku using a single *dyno*[2] which will accept connections with multiple clients via HTTP. The application server will accept JSON data from clients and responds back JSON data as well if necessary. Django's built in User Authentication library will be used to store user's entity for a better reliability in security.

Security Framework

Use Django's built-in API to retrieve top scores from the database instead of using SQL commands to retrieve the information.
This enables us to not enter SQL commands which attackers can use to commence SQL injection attacks to corrupt the users data that we have.

Single Player Data Details

We will store the single player's information within the player's phone as text file so that the player does not have to connect to the Internet to play the game and update their score.
The single player's high score (SPScore) will be uploaded onto the server when the player successfully logs into his account on Multiplayer mode.

- SPScore - an integer for storing its highest Single Player score
- SPMoney - an integer for storing in-game money usable in Single Player mode (for future iteration)
- SPItem - powerups usable only in Single Player mode (for future iteration)

[2] Dyno on Heroku, https://devcenter.heroku.com/articles/dynos

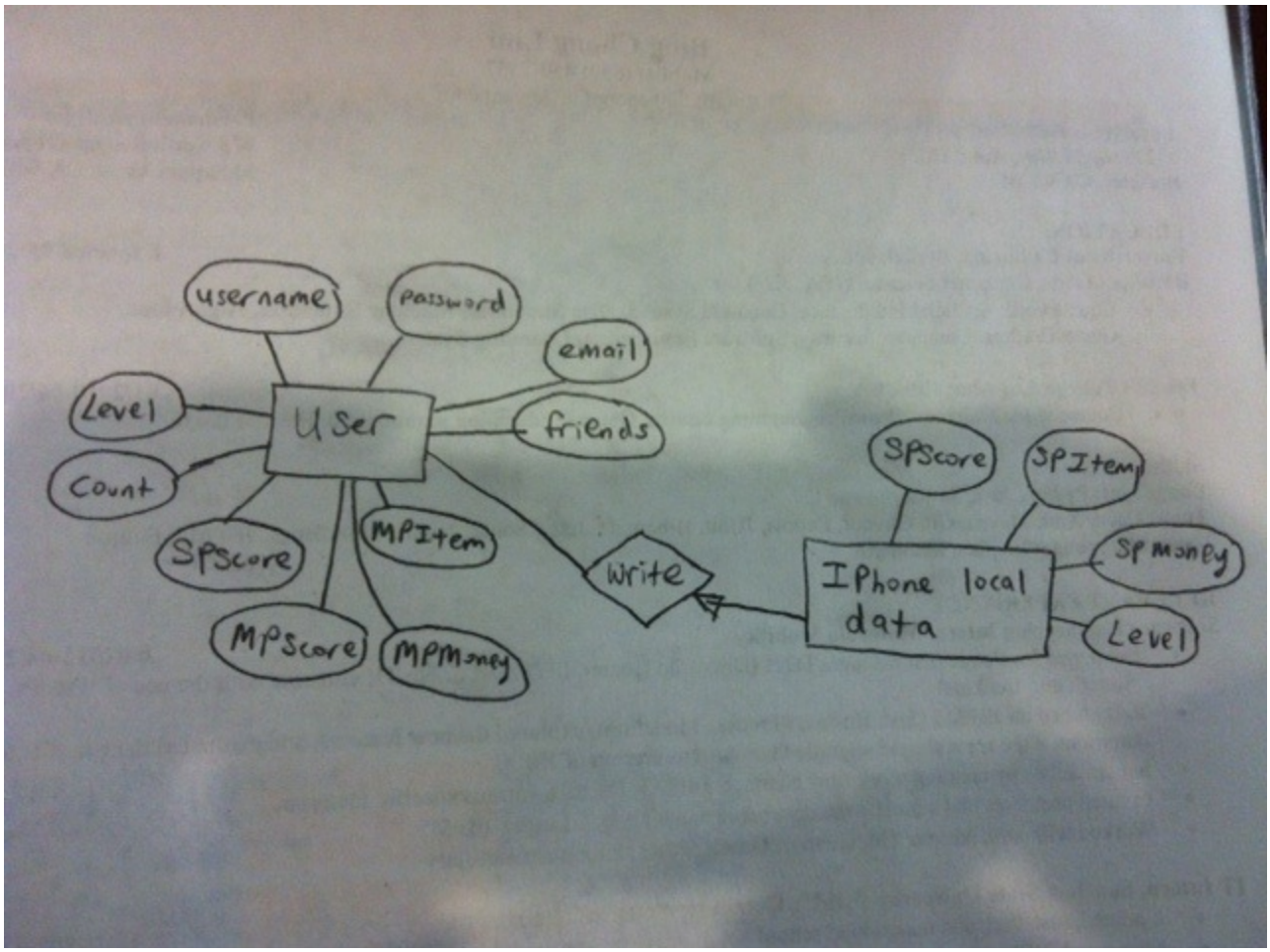
Database Details

The relational database will be a PostgreSQL DBMS. It will store each of the following tables:

Users Table:

- Username - a four to fifteen alphanumeric letters as the primary key (not NULL)
- Password - a four to fifteen alphanumeric letters (not NULL)
- Email - an email string format (not NULL)
- Level - an integer for the highest level the user has achieved
- Count - an integer for counting how many times the user has ever logged in
- SPScore - an integer for storing its highest Single Player score
- MPScore - (not yet implemented)
- MPItem - (note yet implemented)
- MPMoney - (not yet implemented)
- Friends - (not yet implemented)
- ...

ER Diagram



There is one user table in the database that stores all the important information.
The only time when the Iphone's local data has to write data to the database is when the player logs into his account, his single player's high score will be updated within the database.
When the player is playing Single Player mode, his game's information will be stored in a text file locally. (Might use built-in API to do the storing in future iteration)

Invariants

Username
- Cannot be NULL
- Need to be longer than 4 characters
- Less than 16 characters

Password
- Cannot be NULL
- Need to be longer than 4 characters
- Less than 16 characters

Email
- Has to be a valid email
- Account has to be verified from email before able to play Multiplayer mode

Level
- Has to start from 1
- Cannot be less than 1

Count
- Start from 0
- Cannot be less than 1

SPScore
- Has to be player's highest score in Single Player mode

Algorithm

At the start of the game, 3 random words are chosen for the players to construct by the end of the round. The letters that form up the 3 random words are ensured to appear during the phase of the game. (Example: bee, fur, and hive. These 3 words contain 1 'b', 3 'e', 1 'f', 1 'h', 1 'r', 1 'u', and 1 'v'.) These characters are stored into an array and randomly generated characters are also added into this array. (For a 30 seconds game, if each mole will appear for a period of 2 second, we will fill up the array to contain (game_time/duration_of_mole) 30/2 characters. This is a lower bound, assuming that for every period of time that a mole appears, only 1 mole will pop up even though multiple moles are allowed to appear at the same time. The characters in the array are shuffled and when the moles pop up, they will choose a character from the front of the array. The chosen character will then be removed from the array. By the time the array is empty, random characters will be generated for the moles until the end of that particular round.

Protocols

HTTP

HTTP will be used for all client-server communication. All data will be transferred in plaintext

(including user's identity).

JSON

All data transferred between client and server will be in JSON data.

Statistic Page - for Single Player

This Page will show the user's end score and other actions that the user can take after finishing the game. Upon reaching this page, a local file will be read, converted to NSDictionary and checked whether the current score is eligible to be placed in the top 10 local score. If yes, then delete the lowest score and rearrange the position of the score which will then be written back to the file. If not, then it will do nothing. The format of the file will be XML with key as ranking and value as the score.

Score Page

This page will contain 3 tabs: Single Player local top 10 scores, Single Player online top 10 scores and Multiplayer top 10 scores. The last two will require a login through the Login Page. For the initial iteration, only the first two are available and the highest score will taken from the maximum score of any level of difficulty. The difference between SP local and online is that, 10 highest score of the user will be displayed for SP local and only the max of the SP local will be displayed in SP online (assuming the user is in the top 10). The remaining 9 will be filled by other users who are within the top 10 position.

11. Multiplayer Page

This Page will be implemented in future iteration.

12. Shop Page

This Page will be implemented in future iteration.

Client-Server Messages

Action	Client Request	Description	Server Response	Description
Login	username	contains the username	errCode: SUCCESS(1), level, SPScore	username matches with the database and password uses some authentication API
			errCode: ERROR(-1)	invalid Username/Password combination
			errCode: ERROR(-2)	username/Password must be between 4 to 16 letters
After successful login (automatic)	username, local highest score, local level progress	user will upload its highest score to server for updating player's highscore and level progress in database	errCode: SUCCESS(1)	successfully process the score
Forgot Password	email	reset password	errCode: SUCCESS(1)	resets the user’s password, by asking the user to create a different password
			errCode: ERROR(-3)	email field cannot be empty
			errCode: ERROR(-4)	email does not exist in database
Register	username,	contains	errCode:	all three fields are valid

	password, email	username and email text for new user	SUCCESS(1), level	
			errCode: ERROR(-5)	username is already exist
			errCode: ERROR(-6)	username/Password must be between 4 to 16 letters
			errCode: ERROR(-7)	email is not in the right format
Scores	username	contains the current user's username	usernames: values	return a key-value pair with the key as the username and value as the scores

Local XML-format File Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="1.0">
<dict>
   <key>Level</key>
        <integer>3</integer>
   <key>Scores</key>
   <array>
       <integer>1699</integer>
       <integer>1300</integer>
      <integer>1200</integer>
      <integer>1100</integer>
      <integer>1000</integer>
      <integer>1000</integer>
      <integer>1000</integer>
      <integer>1000</integer>
      <integer>1000</integer>
      <integer>1000</integer>
   </array>
</dict>
```

```
</plist>
```

Future Iterations

- Multiplayer mode
- Shops and power-ups
- Penalties for discarding characters
- Different moles (Ex: Moles that wear helmets need to tap twice instead of once)
- Different layouts of the holes
- Tapping on a mole holding a bomb, will make the player lose 3 characters that he is holding onto
- ELO[3] rating system to match user with another “same-level” user when playing Multiplayer with random user

[3] ELO rating system, http://en.wikipedia.org/wiki/Elo_rating_system

Implementation Plan

Application and Database

Backend and server-side methods: **Wenhao and Pei Jun**

- Database schema
- MTV Implementation:
    - Model:
        - requestPassword()
        - register()
        - getScore()
        - login()
    - View:
        - index() // parsing client's request and redirecting to appropriate function call in Model

This will be the whole implementation of the server-side. Login, register, retrieving top scores will be dependable on a running server which will access the database for data.

Frontend Application

Login Page: **Charls and Bing**
Home Page: **Charls and Bing**
Request Password Page: **Charls and Bing**
Forgot Password Page: **Charls and Bing**
Register Page: **Charls and Bing**
Help Page: **Charls and Bing**
Level Select Page: **Denny**
Pause Page: **Denny**
Statistic Page: **Denny**
Stage Page: **Denny, Charls and Bing**

The programming of the frontend app involves creating Pages (i.e. buttons, interface, text field) and interactions between buttons such as opening connection to server as well as implementing the necessary functions of the Pages.

Frontend UI Animations and Graphics

Animations and graphics: **Edward**

The art design of the frontend app. It will cover the graphics of each Pages for a better UI/UX by providing rich images and animations.

Testing Plan

Overview

We are implementing 3 different kinds of testing. First, we have unit testing for backend (Django) to ensure that the function calls and methods work correctly. Second, we have the testing for the frontend (iOS) to ensure that the interfaces are desgined correctly, the buttons link to the correct pages, and the views do not show any overlapping of buttons etc. Lastly, we have the integration test to test both the frontend and the backend after integrating both segments together.

Backend Testing

We will write some unit tests within the backend to test each method and functionality (such as getting the top 10 scores for Single Player mode. We can have unit tests that run these methods individually and check if we get the expected output. (For example: for a unit test to get the top 10 scores for Single Player mode, we can create a new table that holds 15 different scores and run that method on the new table we created. Since we created that new table, we know what is the top 10 scores. Therefore, we can compare the output we get from running that particular method with the correct output that we expect. We can use assert statements to ensure that the method is producing the correct results.)
We will write similar unit tests for every method we have in the backend and run all the unit tests occasionally to ensure that there are no unexpected results.

Frontend Testing

All of us will run the simulator and click on the different buttons that we implemented such as the ‘login’ button or ‘forget password’ button. We have to ensure that every button links us to the correct subsequent view. We will also try to enter both correct and invalid inputs for different fields to ensure that they generate the expected results or error message. (For example: entering a non-existing username and trying to log in will generate an error message)
We will also ask some friends and relatives to play around with the simulator and inform us if there is any weird output or if the simulator crashes.

Integrated Testing

After combining both frontend and backend, we will ask friends and relatives to test out the interface and play a few rounds of the game. Since they are new to the game, they can provide valuable feedbacks on how we can improve the game and interface. Also, they might enter some inputs that we would never have thought of, which might crash the application. This can lead us to some end-cases that we might overlook and resolve the problem before releasing the product.