University of British Columbia

Computer Science Honours Thesis

COSC 449 201

Multiplayer Math on the go with Factor Friends

Author:
Paul Moore

Supervisor:
Dr. Ramon Lawrence

April 2013

Chapter 1

Abstract

Smartphones and other internet capable mobile devices are booming in popularity. Pocket-sized computers are now another welcome distraction to students, both in and out of the classroom. Unfortunately, Math is not becoming any more fun to learn. The solution isn't less technology in the classroom, but more of it. Mobile devices have proven to be great tools for entertainment and social networking, but now it's education's turn. I have created an iPhone Application which applies elements of game theory in a game designed to teach arithmetic. *Factor Friends* makes learning core Math and Computer Science concepts an engaging, multiplayer competition. By integrating social networking, *Factor Friends* becomes a shared learning experience for peers. The game also pairs opponents together who best match each other skills, which provides a natural learning curve and level progression. Gamification is used to not only turn Math into a game, but make it an integral part of the experience while keeping it fun. Concepts such as precedence, association, and reduction are the keys to earning more points, achievements, and even user created content.

Chapter 2

Acknowledgements

I would first like to thank my supervisor, Dr. Ramon Lawrence, for his guidance and willingness to help me with this project. Secondly, this game was made possible by the talented artist Morgan Long[1]. Finally, my friend and family's criticism and role as play testers enabled Factor Friends to be more than it could have ever been with my ideas alone.

[1] View more of Morgan's artwork on her website: `http://artworkofmorganlong.com`

Table of Contents

1 Abstract **i**

2 Acknowledgements **ii**

List of Figures **iv**

List of Listings **v**

3 Factor Friends, the Game **1**
3.1 What is Factor Friends? 1
3.1.1 High Level Concept 1
3.1.2 Target Audience 2
3.2 Factor Friends Game Design 2
3.2.1 Core Concepts 2
3.2.1.1 Learning Targets 2
3.2.1.2 Science Toolbox 4
3.2.2 The Secret Number Game 4
3.2.3 Matches 5
3.2.4 Meta Game 5
3.2.4.1 IQ Points 5
3.2.4.2 Crafting 9
3.3 Accounts 12

4 Technology Overview **14**
4.1 Design Goals 14
4.2 An Overview of the Stack 15
4.2.1 The back-end Stack 15
4.2.1.1 API Server 15

4.2.1.2 Database 22
4.2.1.3 Hosting 22
4.2.1.4 Deployment 22
4.2.1.5 Scaling Out 25
4.2.1.6 Vertical Scaling 25
4.2.1.7 Horizontal Scaling 26
4.2.2 front-end Stack 30
4.2.2.1 Cocos2d 30
4.2.2.2 CocosBuilder 31
4.3 REST API 32
4.3.1 Authentication 32
4.3.2 Request and Response Structure 33
4.4 Client-side Networking 35
4.4.1 Airtower 35
4.4.2 Secure Socket Layer 38
4.5 Implementing Game Play 39
4.5.1 Evaluating Equations 39
4.5.1.1 The bet.coffee Infix Equation Evaluator for CoffeeScript 39
4.5.1.2 API 40
4.5.1.3 The Shunting-Yard Algorithm 40
4.5.1.4 Custom Operators 43
4.5.1.5 Custom Functions 44
4.5.1.6 Crossplatform Implementation 44
4.5.1.7 Client Wrapper 45
4.5.2 Game Messages 46
4.5.2.1 Sending a Message 48
4.5.2.2 Receiving a Message 49
4.5.2.3 Long Polling 50
4.5.2.4 Correctness 50
4.5.2.5 Client-side Game Message Handling 50
4.6 Additional Resources 51

5 Conclusions **52**

Bibliography **53**

List of Figures

3.1 The Factor Friends login screen. 3
3.2 Player 1, creating a puzzle for the opponent. 6
3.3 The menu scene shows you all of your active games and their status. 7
3.4 The finish screen, where each round is reviewed and points are awarded to each player. 8
3.5 The player's initially empty crafting table. 10
3.6 The player adds items to the crafting table to create a formula. 11
3.7 Visualizing user flow through Factor Friends 13

4.1 Version 1.0.0 of the API responds with text/plain 20
4.2 Version 2.0.0 requires the client to accept JSON, but knows to send the error in plain text 21
4.3 Version 2.0.0 responds with JSON, and can accept either /ping or /test as a route 21
4.4 AWS EC2 account running the Factor Friends API Server from an Ubuntu image 23
4.5 AWS EC2 account running the Factor Friends API Server from an Ubuntu image 26
4.6 Additional volumes can be mounted to instances to both increase disk space and enable data to migrate between them. . 27
4.7 CocosBuilder streamlines game development even further with a visual editor. 31

Listings

4.1 A small CoffeScript example 17
4.2 The equivilant compiled JavaScript code 17
4.3 A simple REST web service using Restify 19
4.4 The git post-receive script for version controlled deployment . 23
4.5 Bootstraping the web service to launch worker nodes using the cluster API 25
4.6 Shared code for node1 and node2 27
4.7 Code for thread1 27
4.8 Code for thread2 27
4.9 Code for node1 28
4.10 Code for node2 28
4.11 Obtaining RedisDC 28
4.12 Code for node1 using RedisDC 28
4.13 Code for node2 using RedisDC 29
4.14 Server-side impementation of the authentication scheme 32
4.15 Example HTTP request made to the Factor Friends API web service 33
4.16 The server selects an appropriate Request Handler based on the type field 35
4.17 Example AirtowerRequest implementation 36
4.18 Example AirtowerResponse implementation 36
4.19 Making a request to the web service using the Airtower 37
4.20 Allowing self-signed SSL certificates originating from factor-friends.com in DEBUG mode 38
4.21 Input to the bet.coffee library 40
4.22 Evaluating an infix equation asynchronously using the bet.coffee library 40
4.23 Synchronous API of the bet.coffee library 40

4.24 Pseudocode for the Shunting-Yard Algorithm 41
4.25 Evaluating equations in Reverse Polish Notation 42
4.26 Creating or redefining an operator using bet.coffee 43
4.27 Creating or redefining a function using bet.coffee 44
4.28 Setting up bet.coffee in a web view 45
4.29 Interfacing to bet.coffee 45
4.30 Evaluating an equation from Objective-C 46
4.31 The Lua script to add an event to a player's message queue . . 48
4.32 Pseudocode for receiving new messages with a **listen** request . 49
4.33 The menu scene listining for game messages 51

Chapter 3

Factor Friends, the Game

3.1 What is Factor Friends?

Factor Friends is a multiplayer, mobile, educational game. Unlike a conventional educational game, which takes a game and adds math on top of it, math in Factor Friends *is* the game. Factor Friends uses the inherent game like properties of math to make an interesting, casual game for handheld devices.

The goal of Factor Friends is to create a fun, social, and educational experience for all ages. It is not, however, designed to teach a curriculum. Instead, Factor Friends aims to show how math can be fun, and how games can be used to teach new ways of learning.

3.1.1 High Level Concept

The game is intended to be a single release onto the Apple App Store. Additional content may become available in future App updates. The business

model will be based primarily around the monetization of purchasable gameplay elements via In-App Purchases. The initial App cost will be low, or free, to reduce the barrier of entry.

3.1.2 Target Audience

The App will not be targeted at a specific demographic or age group. However, an effort will be made to keep the App accessible to a younger audience, primarily between the ages of 6 and 10.

Recent studies show that tablet and smartphone use is on the rise among children, where 77% are playing games and 57% are using them for educational purposes [1]. Since Factor Friends is an educational game, these trends justify the effort into making the App available to a younger age range.

3.2 Factor Friends Game Design

3.2.1 Core Concepts

3.2.1.1 Learning Targets

Each game and educational feature in Factor Friends will be based one of the following concepts:

Game Concepts

1. Mathematical and Computer Science operators and functions.

    - The player is able to learn, use, share, and craft operators from math and common programming languages.
    - The game progresses from basic operators (addition, subtraction, multiplication, division) to more complicated ones (exponent, modulus, square root).
    - Computer programming operators are also available (pre/post increment, bit-shifting).
    - More advanced players can also use logical and algebraic operators (AND, OR, etc.).
    - In addition to operators, functions can also be used, such as floor, ceiling, and clamp.

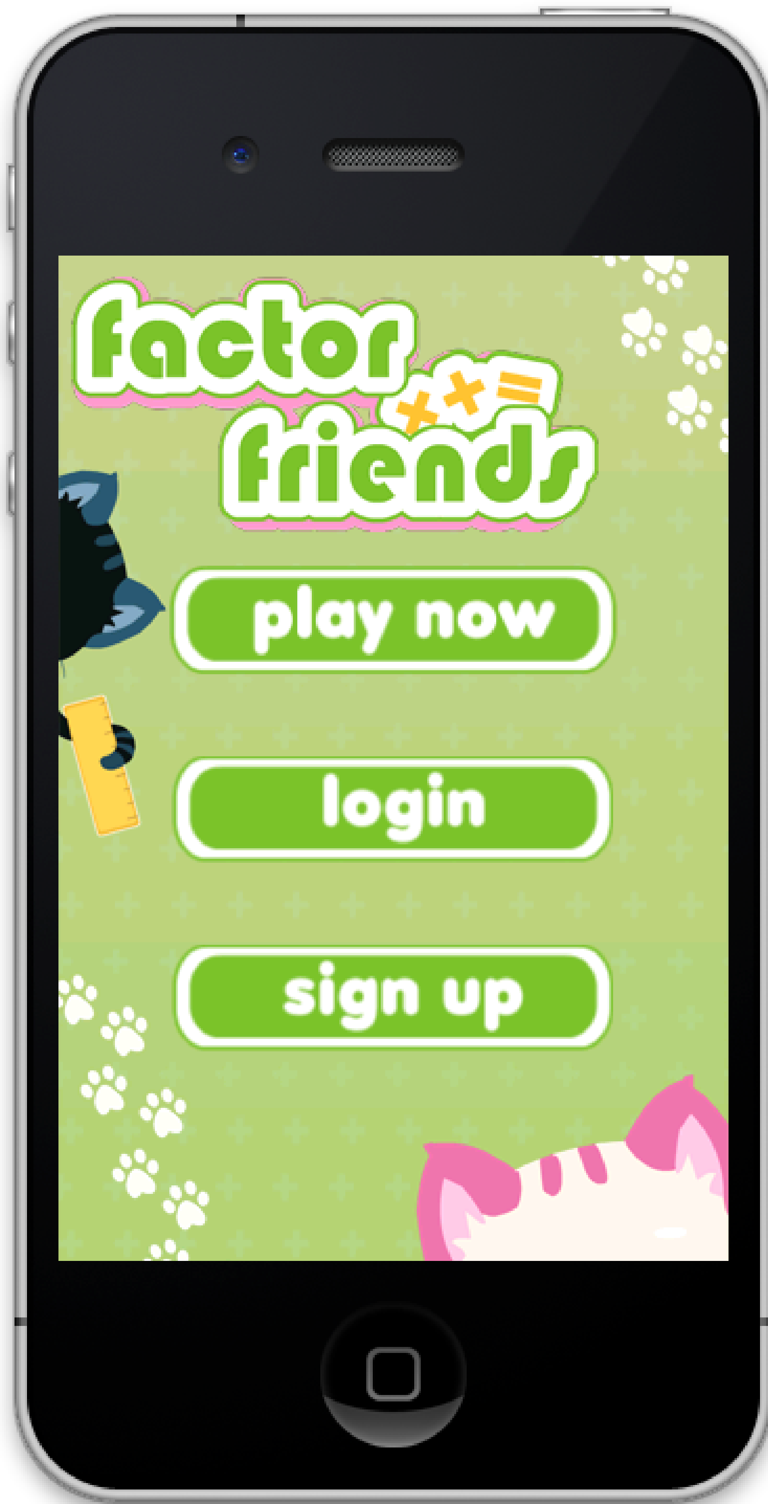

Figure 3.1: The Factor Friends login screen.

2. Problem solving.
    - Players are introduced to problem solving by having to figure out other player's puzzles.
    - Puzzles are mathematical equations that are created by operators, as discussed above.
    - Players are involved in the puzzle creating process, as well as the puzzle solving process.

3.2.1.2 Science Toolbox

The Science Toolbox is the inventory system for Factor Friends. Each player starts with a Toolbox with a few basic operators. As players progress through the game, they earn more things to add to their Toolbox through various ways. The toolbox is an essential part to the player's experience, as it dictates what the player can and cannot use during a game.

3.2.2 The Secret Number Game

The core game behind Factor Friends is a fairly simple *secret number game*. Instead of solving predefined problems, the player participates in the puzzle creating process, and uses those puzzles to challenge friends online. Below is how a typical match between two player's would play out.

Game Flow

1. Player 1 (P1), and Player 2 (P2) enter a game match together.
2. It is P1's turn first.
3. A random selection of numbers and operators are selected from P1's Toolbox.
4. P1, using that random selection, creates a Puzzle using all, or some of those numbers and operators.
5. The answer to the Puzzle (limited to a numerical value) is sent to P2.
6. P2 then uses the same Toolbox selection in an attempt to solve the Puzzle.

7. Points are awarded based on how close P2 came to figuring out what P1's original equation was.

3.2.3 Matches

A Match in Factor Friends is two game rounds between two players. A round is when one player creates a puzzle, and the other player solves that puzzle. Therefore, a match can be described by this sequence:

Game Rounds

1. P1 creates a puzzle for P2.
2. P2 solves the puzzle P1 created.
3. P2 creates a puzzle for P1.
4. P1 solves the puzzle P2 created.

Points are awarded only at the end of a match. Thus, equal opportunity is provided for each player to score points if one player decides to quit.

A match can be instigated by either player, however, only one match can be in play at a time between two players. This makes it easier for players to organize their matches (one match per friend at a time), and avoids game creation spamming.

3.2.4 Meta Game

A meta game is the game beyond the core game experience. In Factor Friends, the core game is the secret number game, and the meta game (discussed below) is the reason for players to continue playing the core game.

3.2.4.1 IQ Points

IQ Points is the currency system behind Factor Friends. At the end of a match, players are rewarded IQ based on how well they did. These points are persistent across matches, and can be accumulated by the player.

Once obtained, IQ can be spent in the following ways:

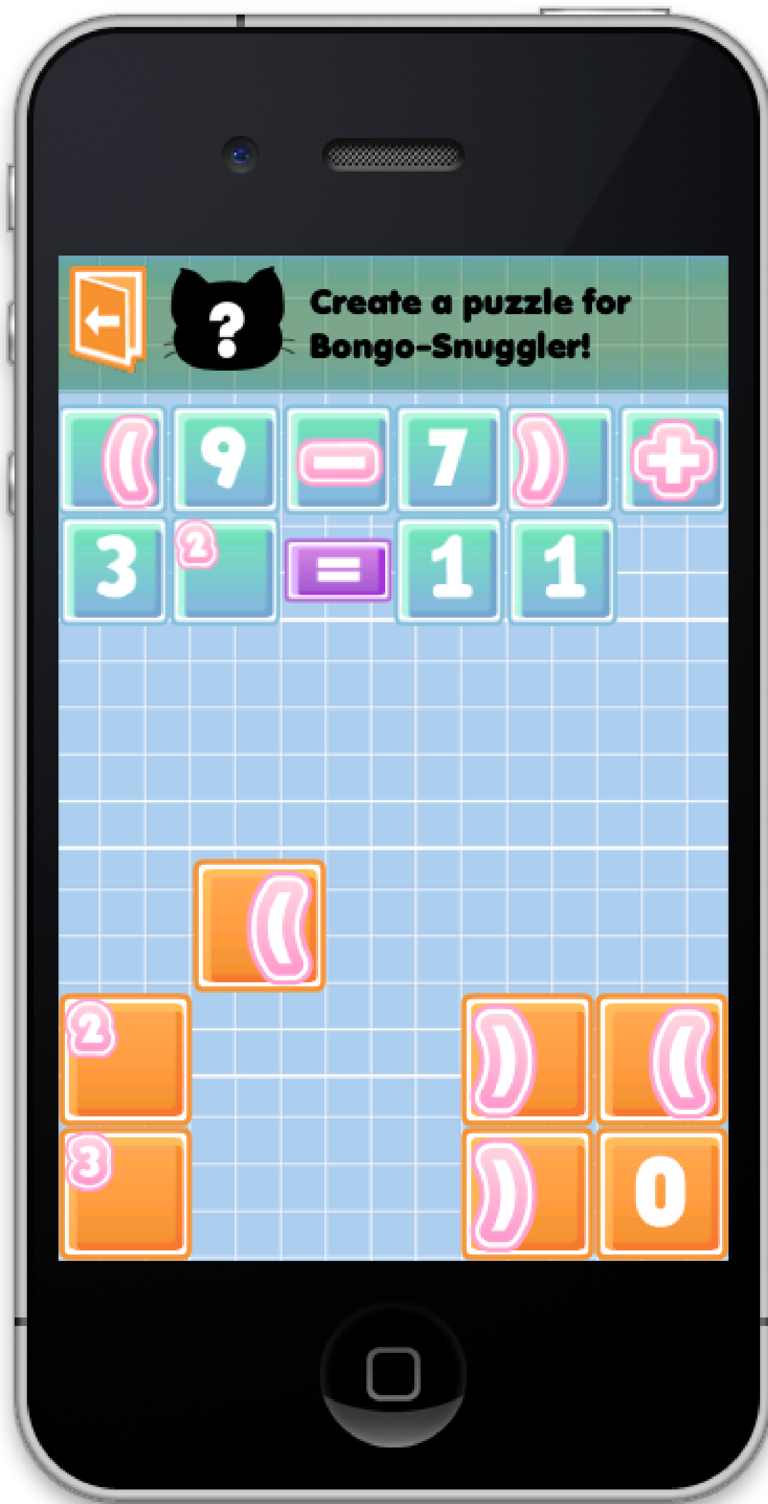

Figure 3.2: Player 1, creating a puzzle for the opponent.

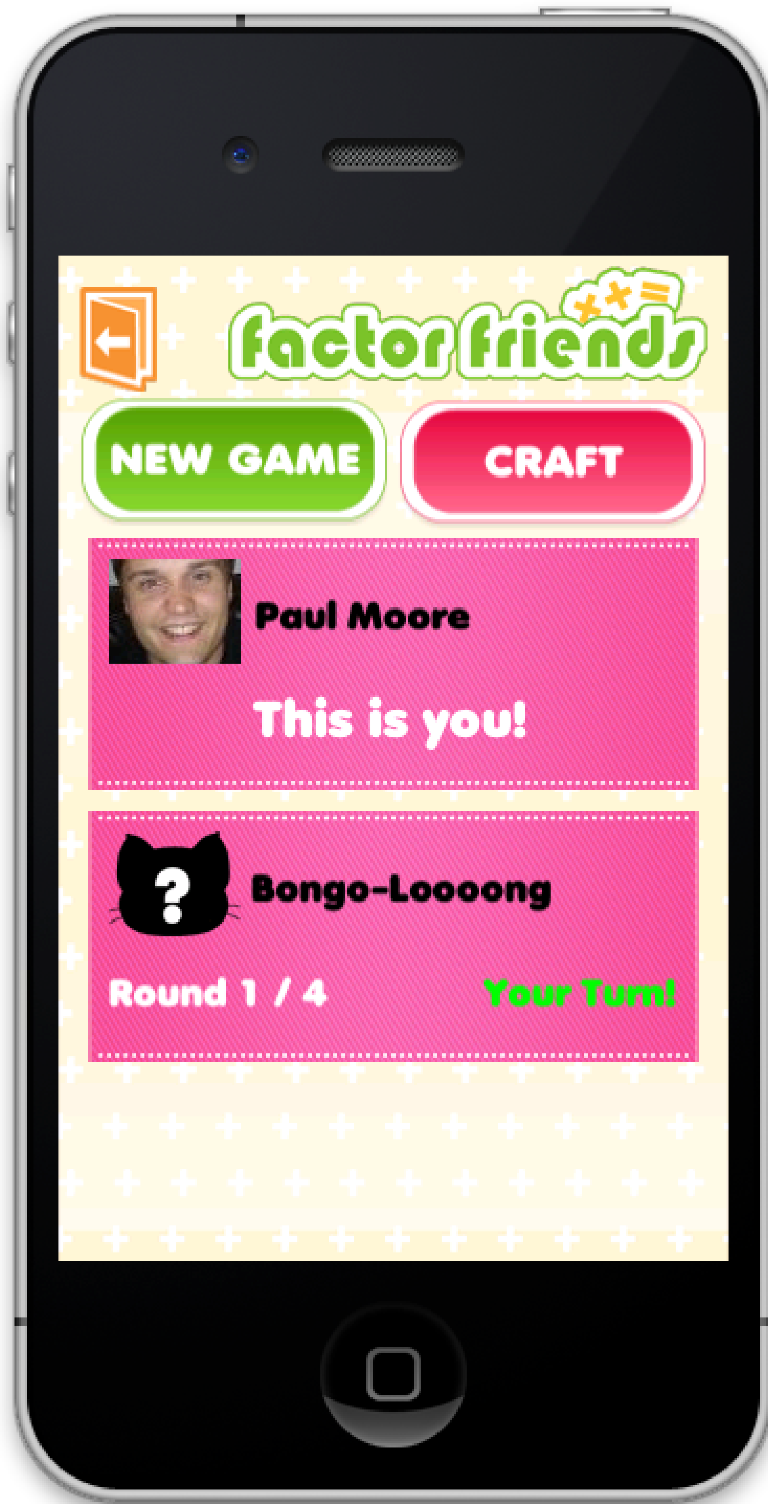

Figure 3.3: The menu scene shows you all of your active games and their status.

Figure 3.4: The finish screen, where each round is reviewed and points are awarded to each player.

Toolbox Upgrades - The Toolbox Store A player's Toolbox can be expanded with new operators, functions, and numbers. Using an in game store, a player can exchange IQ Points for these things. Each new purchase is followed by a small, micro tutorial of how to use it.

When a player gets a new addition to his or her Toolbox, it becomes immediately available to use the next time the player enters a new round. For immediate positive feedback, new purchases are always a part of the next round the player controls; this way they get to try out their new skills quickly and be satisfied with them.

A Note on Microtransactions - The IQ Point Store Though not the main focus of the project, the game may be monetized by offering In-App Purchases for more IQ Points. These purchases allow the player to 'top up' their IQ, allowing them to reach their goal of buying a particular skill sooner. This sort of model monetizes on a player's impatience.

This store would be separate, but directly accessible, from the Toolbox store. The IQ store currently has the following available purchases:

IQ Store In-App Purchases

- 10 IQ Points: \$0.99
- 30 IQ Points: \$1.99 (10 IQ free!)
- 50 IQ Points: \$2.99 (20 IQ free!)

3.2.4.2 Crafting

The Second component to the meta game is a Crafting component. Crafting in Factor Friends works by taking existing items from your Toolbox inventory, and using them to create new items for yourself.

For example: Assume a player has the addition operator, +, as part of his or her Toolbox. The player's crafting table will initially look empty, like this:

The player can then drag the addition operator, and other symbols into the crafting space. In this example, the player adds one to a number, which results in the increment operator, ++.

Through crafting, the player has found a new recipe of sorts for the equation $x = x + 1$. The player can now use this new operator as part of his or her Toolbox. The original operator, +, still remains in the player's toolbox.

Figure 3.5: The player's initially empty crafting table.

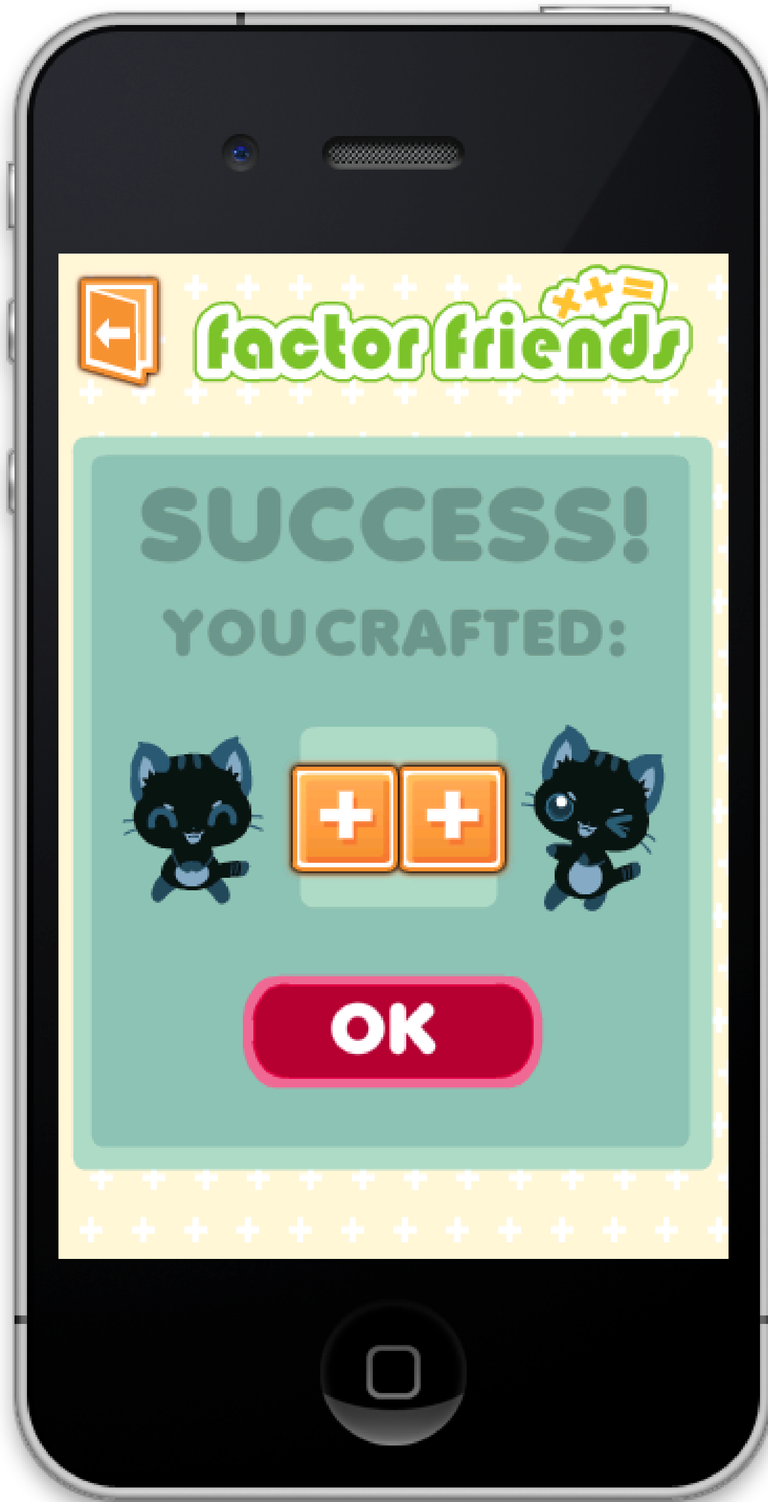

Figure 3.6: The player adds items to the crafting table to create a formula.

Crafting encourages experimentation and rewards the player with discovery. The goal is to have the player build intuition about math and to help them see similarities which may help them in the puzzles.

3.3 Accounts

To play Factor Friends, each user must first create an account. There are currently two options for creating an account.

If you just want to try the game without having to sign in or give any personal information, you can play as an anonymous user. The disadvantage to playing anonymously is that your information is not saved when you log out, and you cannot use the IQ Store.

The other option is to sign in with Facebook. This method allows you to easily connect with your friends that also play Factor Friends. Using this method, your data is saved permanently. In addition, you can play Factor Friends on multiple devices using the same account.

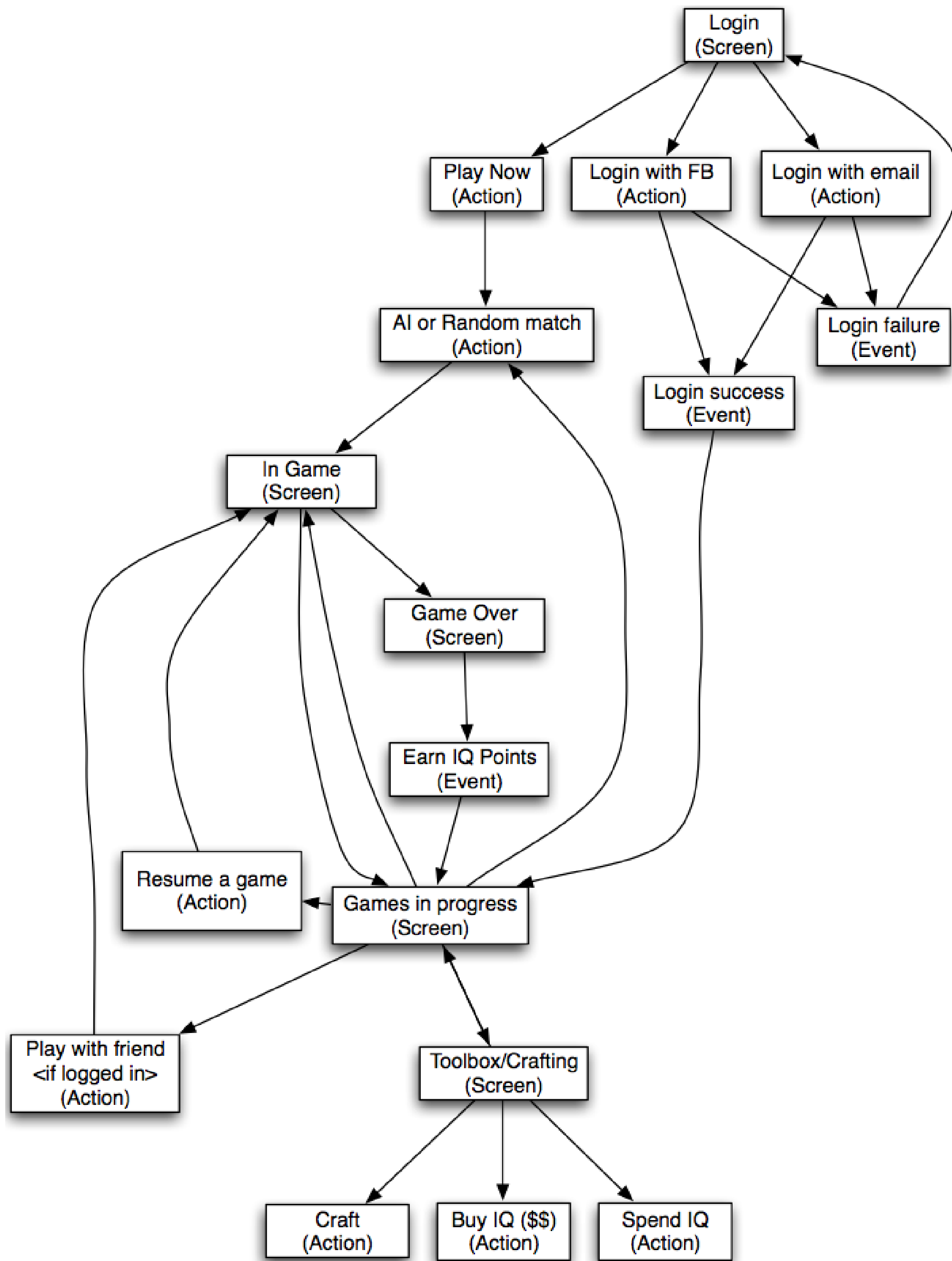

Figure 3.7: Visualizing user flow through Factor Friends

Chapter 4

Technology Overview

The architecture of Factor Friends is broken into two major components: the forward facing client and the back-end stack. This section will discuss the major components of how Factor Friends was designed and implemented, and what technologies were used.

4.1 Design Goals

As the game mechanics of Factor Friends changed often, and with the possibility of a sequel in the future, the system needed to be robust and adaptive to change. This led to decisions that favoured quick development and fast turnaround. Below are the key attributes I was looking for when I choose which technologies I was going to use on the back-end:

Desired Attributes

1. Fast development of new features
2. Easy to deploy and test
3. Client agnostic
4. Built upon popular open standards

The first two points are self-evident and are desired properties of any technology. However, since I was the sole developer on both the client and server, these points became increasingly important to reduce workload and focus on more critical pieces of the application. Furthermore, an integrated

build system was also needed to reduce the time between compiling, deploying, testing, and debugging new code.

Additionally, the server should be independent of the client application. This is to enable the future development of a client or service which is created using a different platform than the original. For instance, an HTML5 version of the game may be a possibility. Or, should the API server expand into a platform, developers may wish to interface with it using a different technology.

Finally, with the possibility of additional clients or services that need to interface with the API server, I wanted something that was easy to communicate with in any language. Creating a custom protocol creates more opportunity for bugs, and the unfamiliarity will reduce developer adoption, and my will to do any further work with the project. This lead to the decision of adopting a *RESTful*-like approach which will be discussed later.

4.2 An Overview of the Stack

4.2.1 The back-end Stack

We can divide the back-end architecture further into three separate subcomponents: the API Server, the Database, and the Hosting Platform.

4.2.1.1 API Server

The API Server is responsible for handling all requests from clients to perform virtually any action. For instance, a user who uses a new device with his or her account will require all account and game information he or she has accumulated. It is the server's responsibility to fetch and serve this information to the client. In addition, the API Server acts as an *authoritative multiplayer game server*. This means the server also oversees all activity that goes on between games. For every move made, the server validates it and records it in the Database before notifying the player's opponent. This has the advantage over a peer-to-peer or non-authoritative system that needs to handle the case when two clients are in conflict over an action.

Build on Node.js As for the platform, **Node.js**[1] was chosen for the API Server. Node, at its core, is essentially just Google's V8 JavaScript engine[2] with an event loop (using the **libenv**[3] library). In addition, Node extends JavaScript by adding many rich APIs such as networking, cryptography, and filesystem operations implemented natively in C++ for performance. JavaScript own it's own has a very bland API, which is why it was chosen for Node to begin with according to Node's creator [2].

This has two interesting consequences. First, the server can be written in a lightweight, dynamic scripting language. Secondly, unlike a traditional web server, Node is not simply a *directory of files*. What this means is that an HTTP request to:

```
GET /index.html
```

does not have to resolve to an actually HTML file residing in the website's document root. Instead, we can decide to respond to any request however we want. While this functionality is useful for view based applications, writing REST APIs like this becomes tedious. This is evident when we need to implement a rich API for requests such as:

```
DELETE /user/123/cart/item?count=2
```

Finally, Node has a very strong development community for writing third party packages. For these reasons Node.js was chosen to quickly develop a fast, adaptive, and portable API server.

CoffeeScript While Node has the advantage of offering fast development via a dynamic scripting language, it is, unfortunately, JavaScript. JavaScript is known for being fairly verbose in comparison to other languages such as Python or Lua. It adopts much of its syntax from languages such as C and Java instead. In addition, JavaScript suffers scope issues with the *this* variable and offers some features of, but does not fully implement, the OOP and Functional paradigms.

[1]Node.js: `http://nodejs.org/`
[2]Google V8: `https://code.google.com/p/v8/`
[3]libenv: `http://software.schmorp.de/pkg/libev.html`

In recent years, many compile-to-JavaScript languages have been born to avoid the bad parts of JavaScript. One of the most popular of these is called **CoffeeScript**[4], and is the language I choose to develop the API server in. CoffeeScript offers a less verbose and more Functional looking syntax. At the same time, it offers more OOP features and syntactical sugar. Observe:

Listing 4.1: A small CoffeScript example

```
1 unravel = (obj, arr = []) ->
2   arr.push k, v for k, v of obj
3   arr
4 ravel = (arr, obj = {}) ->
5   obj[n] = arr[i + 1] for n, i in arr when i % 2 is 0
6   obj
7 obj =
8   name: 'Paul'
9   degree: 'Computer Science'
10 arr = unravel obj, ['school', 'UBC']
11 # ["name", "Paul", "degree", "Computer Science", "school", "
     UBC"]
12 console.log arr
13 # {"name": "Paul", "degree": "Computer Science", "school": "
     UBC"}
14 console.log ravel arr
```

Listing 4.2: The equivilant compiled JavaScript code

```
1 var unravel = function(obj, arr) {
2   var k, v;
3   if (arr == null) {
4     arr = [];
5   }
6   for (k in obj) {
7     v = obj[k];
8     arr.push(k, v);
9   }
10   return arr;
11 };
12 var ravel = function(arr, obj) {
13   var i, n, _i, _len;
14   if (obj == null) {
15     obj = {};
```

[4]CoffeeScript: http://coffeescript.org

```
16   }
17   for (i = _i = 0, _len = arr.length; _i < _len; i = ++_i) {
18     n = arr[i];
19     if (i % 2 === 0) {
20       obj[n] = arr[i + 1];
21     }
22   }
23   return obj;
24 };
25 var arr, obj;
26 obj = {
27   name: 'Paul',
28   degree: 'Computer Science'
29 };
30 arr = unravel(obj, ['school', 'UBC']);
31 // ["name", "Paul", "degree", "Computer Science", "school", "
     UBC"]
32 console.log(arr);
33 // {"name": "Paul", "degree": "Computer Science", "school": "
     UBC"}
34 console.log(ravel(arr));
```

CoffeeScript improves readability and reduces overall code clutter. As of CoffeeScript 1.6, source maps can be optionally generated. Source maps tell the JavaScript runtime how to display the original CoffeeScript code instead of the compiled JavaScript. This is useful in development mode when if a stack trace is printed when an exception is raised, for example.

RESTful Design The REST (**Re**presentational **S**tate **T**ransfer) architectural style was chosen as the communication model between client and server. More precisely, the API Server is a *RESTful web service*, in that it uses REST style design implemented using HTTP. It should be noted that true REST implementations are *stateless*; that is, no session information is stored on the server. Factor Friend's implementation does abide by this rule, in that the client must re-send authentication information with each request. However, some session data must be stored in the Database for practical reasons, which will become obvious in the next session. Hence, I refer to the Factor Friend's web service as *RESTful-like*.

These technologies were chosen because they are built upon popular open web standards, on of the key design goals. HTTP is supported virtually everywhere, so the opportunity for creating new clients is viable. In addition, the protocol is expressive and flexible enough to handle the complex nature

of a multiplayer game.

The API Server uses the **restify**[5] package for Node. Restify is a framework designed for RESTful web services. It provides facilities to handle content negotiation, versioning, routing, and error handling for an application. Here is example of a simple ping server:

Listing 4.3: A simple REST web service using Restify

```
1 restify = require 'restify'
2 app = restify.createServer name: 'ping-server'
3 app.use restify.acceptParser app.acceptable
4 app.get
5   path: '/ping'
6   version: ['1.0.0', '1.1.0'],
7   (req, res, next) ->
8     res.send 'pong'
9     next()
10 app.get
11   path: /^\/(ping|test)$/
12   version: '2.0.0',
13   (req, res, next) ->
14     if req.accepts 'application/json'
15       res.send response: 'pong'
16     else
17       res.send new restify.WrongAcceptError
18         'You must explicitly accept JSON in version 2!'
19     next()
20 app.listen process.env.PORT, ->
21   app.log.info "Server running on #{process.env.PORT}"
```

In the above example we create a simple server called ”ping-server”. It has a single *route* (a URL which acts as an API end point) to `/ping` which sends a simple ”PONG” response. The server has three versions. In version 1.0.0 and 1.1.0, the response is sent back in plain text.

[5]restify: `http://mcavage.github.io/node-restify/`

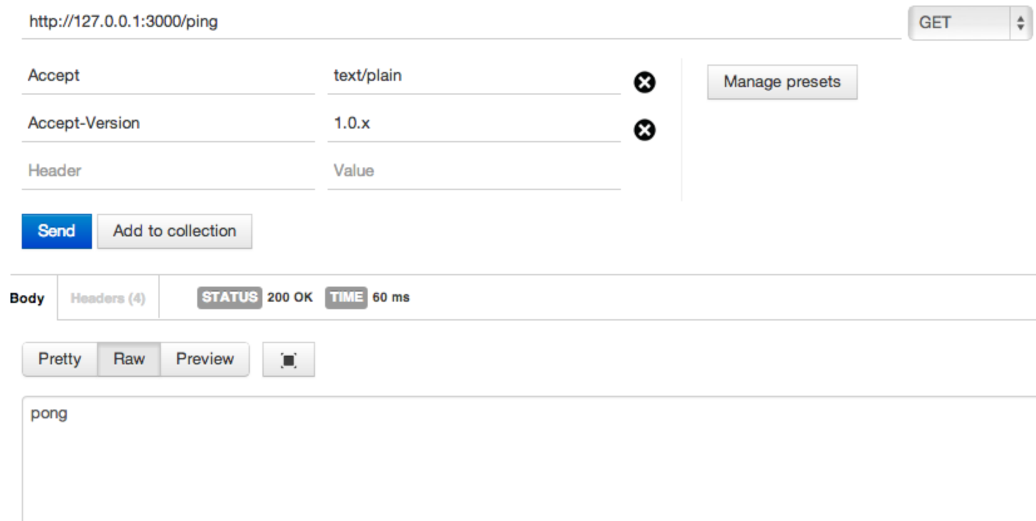

Figure 4.1: Version 1.0.0 of the API responds with text/plain

However, version 2.0.0 of the API sends back a response as JSON, and explicitly requires the client to accept it. Additionally, version 2.0.0 uses a Regular Expression to define its route, which also accepts `/test` in addition to `/ping`.

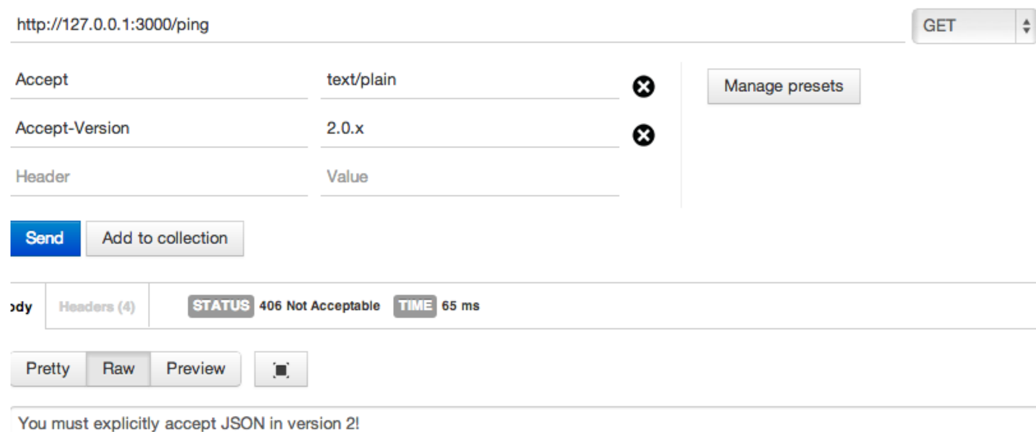

Figure 4.2: Version 2.0.0 requires the client to accept JSON, but knows to send the error in plain text

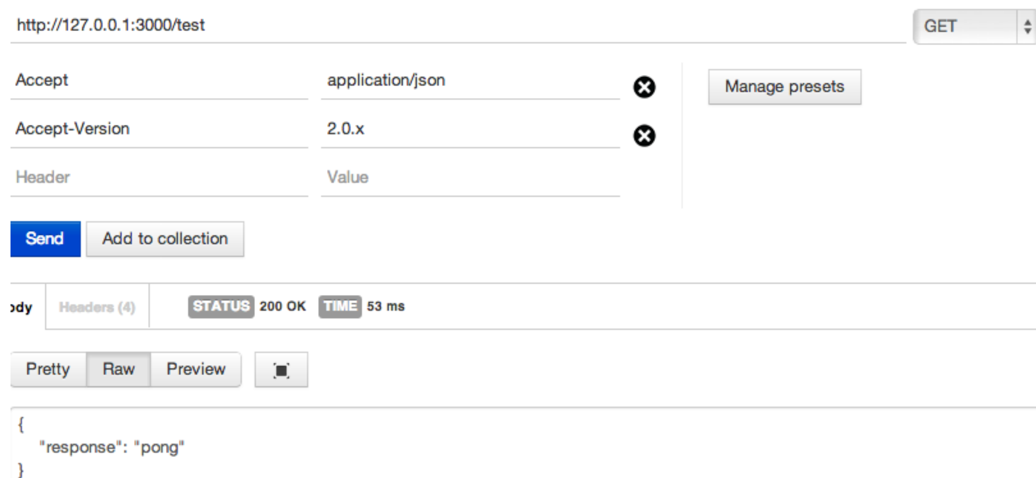

Figure 4.3: Version 2.0.0 responds with JSON, and can accept either /ping or /test as a route

Restify uses the `Accept` and `Accept-Version` headers, among others, to determine the appropriate handler to send the request to.

4.2.1.2 Database

The NOSQL database Redis[6] is used to store non-volatile data. While a traditional SQL database could have been used, Redis offers unique features which benefit the design on the API Server. The publish and subscribe model and the blocking operations of Redis provide concurrency control when the API Server is distributed to multiple nodes. These benefits will become clear in the next section.

As for Redis itself, the API consists of commands to manipulate data structures. Strings, Lists, Hashes, Sets, and Sorted Sets are all supported by Redis. However, there is no concept of a relation. In addition, while Redis does offer some transactional support in a sense that commands can be pipelined together and executed atomically (by virtue of Redis being single-threaded), there are no rollbacks. Hence, the Lua scripting feature of Redis is used to create more durable queries.

4.2.1.3 Hosting

To provide the easiest, most configurable, and least expensive hosting solution, I decide to used Amazon's EC2[7] (**E**lastic **C**ompute **2**). EC2 offers virtualized servers that can be created and destroyed at will. Each instance can be assigned an IP address that you allocate, called an *Elastic IP*.

4.2.1.4 Deployment

Deploying to the production server is done via git. Git allows you to write special scripts, called *hooks*, to be executed after certain events. In this case, we want to perform some work after someone has pushed new code to the server. First, `git-core` was installed and setup on the deploy server. After that, a bare git repository was initialized. Then, a `post-receive` hook was created in the repository's `/hooks` directory. It is as follows:

[6]Redis: `http://redis.io/`

[7]Amazon Web Services: `http://aws.amazon.com/`

	Name	Instance	AMI ID	Root Device	Type	State	Status Checks	Alarm Status	Monitoring
☐	SRD	i-d253dba1	ami-3fec7956	ebs	t1.micro	stopped		none	basic
☑	Factor Friends	i-4e0fc72f	ami-3fec7956	ebs	t1.micro	running	2/2 checks pa	none	basic
☐	Crucible	i-47dacf25	ami-3fec7956	ebs	m1.medium	running	2/2 checks pa	none	basic
☐	OS	i-8c884ae9	ami-3fec7956	ebs	t1.micro	running	2/2 checks pa	none	basic

1 EC2 Instance selected.

EC2 Instance: Factor Friends (i-4e0fc72f)

107.20.149.174

Description | Status Checks | Monitoring | Tags

AMI: ubuntu/images/ebs/ubuntu-precise-12.04-amd64-server-20130124 (ami-3fec7956)

Zone:	us-east-1d	**Alarm Status:**	none
Type:	t1.micro	**Security Groups:**	Factor Friends. view rules
Scheduled Events:	No scheduled events	**State:**	running
VPC ID:	-	**Owner:**	030134338079
Source/Dest. Check:		**Subnet ID:**	-
Placement Group:		**Virtualization:**	paravirtual
RAM Disk ID:	-	**Reservation:**	r-1407e569
Key Pair Name:	FactorFriendsEC2	**Platform:**	-
		Kernel ID:	aki-88aa75e1

Figure 4.4: AWS EC2 account running the Factor Friends API Server from an Ubuntu image

Listing 4.4: The git post-receive script for version controlled deployment

```
1  #!/bin/bash
2  # Post receive script for git deployment.
3  # Factor Friends API Server.
4  # https://github.com/paulmoore/Factor-Friends-API
5  #
6  # To use:
7  # 1. cd /home/git/ff-api.git/hooks
8  # 2. cp ~/post-receive ./post-receive
9  # 3. chmod +x post-receive
10 # 4. chown git:git post-receive # for good measure
11 echo "----------------------------"
12 echo "--      POST-RECEIVE      --"
13 echo "----------------------------"
14 deploy_dir=/var/www/ff-api
15 deploy_branch=live
16 while read oldrev newrev ref
17 do
18   branch=`echo $ref | cut -d/ -f3`
19   echo "Current branch: $branch"
20   if [ "$branch" == "$deploy_branch" ]; then
```

```
21     echo "Stopping server..."
22     (cd $deploy_dir && npm stop)
23     echo "Stopped"
24     echo "Checking out build..."
25     GIT_WORK_TREE=$deploy_dir git checkout live -f
26     echo "Changes pushed live"
27     cd $deploy_dir
28     echo "Building and deploying..."
29     npm install
30     echo "Built"
31     echo "Starting server..."
32     npm start
33     echo "Deployed"
34   else
35     echo "Branch is not live, nothing to do"
36   fi
37 done
38 echo "Done"
39 echo "--------------------------"
```

Post-Receive Hook

1. Check what branch is being pushed to
2. If the branch that is being pushed to is 'live', execute the rest of the script
3. Stop the server if it is currently running
4. Checkout a fresh copy of the server to the deploy directory
5. Using the Node package manager, pull down all dependencies
6. Build the project (compile and lint CoffeeScript)
7. Start the server

On the local development machine, a remote is added to the new git repository.

```
1 $ git remote -v
2 github  git@github.com:paulmoore/Factor-Friends-API.git (fetch
    )
3 github  git@github.com:paulmoore/Factor-Friends-API.git (push)
4 prod  git@factorfriends.com:ff-api.git (fetch)
5 prod  git@factorfriends.com:ff-api.git (push)
```

To deploy, instead of pushing the master branch to the origin server (in this case, GitHub), we push whatever branch we want to deploy to the production server's live branch:

```
1 $ git push prod master:live
```

Using this technique to deploy to the production server is simple and very fast as it is integrated with normal development.

4.2.1.5 Scaling Out

In anticipation that the game will attract thousands of players, or none at all, there needed to be a plan to quickly increase or decrease the capacity of the API Server depending on the need. Both vertical and horizontal scaling techniques were used to reduce headaches later if the game grows in popularity.

4.2.1.6 Vertical Scaling

Using the cluster API . Since Node.js is single threaded, we can increase the load capacity of the server by simply taking advantage of all of the machines cores. The cluster module allows Node to spawn worker threads which share any ports they listen on. The master thread then forwards requests evenly to its workers.

Listing 4.5: Bootstraping the web service to launch worker nodes using the cluster API

```
1 cluster = require 'cluster'
2 if cluster.isMaster
3   if process.env.NODE_ENV is 'development'
4     cpus = parseInt process.env.
          npm_package_config_debugWorkersN
5   else
6     cpus = require('os').cpus().length
7   log = require('./logger').create require('../config').server
        .logName
8   log.info "Master process started with #{cpus} processors"
9   cluster.fork() for i in [1..cpus]
10  cluster.on 'fork', (worker) ->
11    log.info "Worker #{worker.id} forked"
12  cluster.on 'online', (worker) ->
13    log.info "Worker #{worker.id} online"
14  cluster.on 'listening', (worker, address) ->
```

```
15      log.info "Worker #{worker.id} listening on #{address.
          address}:#{address.port}"
16    cluster.on 'exit', (worker) ->
17      if worker.suicide
18        log.info "Worker #{worker.id} committed suicide, not
            restarting"
19      else
20        log.info "Worker #{worker.id} has died, restarting"
21        cluster.fork()
22  else
23    require './worker'
```

The server first determines if it is a worker or master thread. If the current process is the master, it spawns an appropriate amount of workers and listens for any changes to their state. If if is a worker, we setup the API routes and begin listening for HTTP requests as normal. It is also possible for the master to restart one of its workers if it goes down unintentionally.

4.2.1.7 Horizontal Scaling

Amazon Machine Images In addition, you can create custom images from you devices to launch identical VMs. This way, you only need to setup a server once, all others can be duplicated. This is useful if you need to scale out horizontally quickly by creating new VMs. Using Amazon's EC2 Load Balancer automatically distributes traffic across your VMs.

Filter: Owned By Me | All Images | All Platforms | Search AMIs... | 1 to 2 of 2 AMIs

Name	AMI Name	AMI ID	Source	Owner	Visibility	Status	Platform	Root Dev
Crucible	Crucible	ami-d458c2bd	03013433807...	030134338079	Private	available	Other Linux	ebs
Factor Frie...	Factor Friends AMI	ami-4ef66d27	03013433807...	030134338079	Private	available	Other Linux	ebs

Figure 4.5: AWS EC2 account running the Factor Friends API Server from an Ubuntu image

Additional volumes can also be created and mounted to instances. These volumes can be used to store important data, such as database files. In this case, we store the Redis snapshots and AOF files to a separate volume. If something tragic happens to the EC2 instance running the Database, the volume can be mounted to a new instance and the original instance can be disposed.

	Name	Volume ID	Capacity	Volume Type	Snapshot	Created	Zone	State	Alarm Stat	Attachment Information
☑	*empty*	vol-a61fe8d6	8 GiB	standard	snap-bcdff2f2	2013-02-26T09:25:31	us-east-1c	in-use	*none*	i-d253dba1 (SRD):/dev/sda1 (attached)
☑	*empty*	vol-e449a097	8 GiB	standard	snap-bcdff2f2	2013-03-21T05:22:12	us-east-1d	in-use	*none*	i-4e0fc72f (Factor Friends):/dev/sda1 (attache
☐	*empty*	vol-e38e0690	8 GiB	standard	snap-bcdff2f2	2013-03-29T19:19:30	us-east-1c	in-use	*none*	i-47dacf25 (Crucible):/dev/sda1 (attached)
☐	*empty*	vol-e7830b94	10 GiB	standard	--	2013-03-29T19:22:06	us-east-1c	in-use	*none*	i-47dacf25 (Crucible):/dev/sdf (attached)
☐	*empty*	vol-594d7d1f	8 GiB	standard	snap-bcdff2f2	2013-04-06T01:02:19	us-east-1d	in-use	*none*	i-8c884ae9 (OS):/dev/sda1 (attached)

Figure 4.6: Additional volumes can be mounted to instances to both increase disk space and enable data to migrate between them.

Redis Distributed Concurrency Distributing the workload across multiple nodes has the disadvantage of being much more difficult to control concurrency. Traditional control structures such as the *semaphore* or *mutex* cannot be used in a traditional sense when critical sections need to be shared across multiple nodes. To illustrate this, here is a classical example of a concurrency problem.

Concurrency problem using threads

1. **a1** on **thread1** must occur before **b2** on **thread2**

2. **b1** on **thread2** must occur before **a2** on **thread1**

A traditional approach would be to use semaphores or some other form of locking mechanism provided by the system. The solution in this case uses the *rendezvous* pattern.

Listing 4.6: Shared code for node1 and node2

```
a1Done = Semaphore(0)
b1Done = Semaphore(0)
```

Listing 4.7: Code for thread1

```
a1
a1Done.signal()
b1Done.wait()
a2
```

Listing 4.8: Code for thread2

```
b1
b1Done.signal()
a1Done.wait()
b2
```

To solve this problem when some or all events occur on a different node, a library was developed to use Redis as a concurrency control structure for multiple nodes. The library, called RedisDC (**R**edis **D**istributed **C**oncurrency)

uses the blocking operations of lists to emulate semaphores. The same concurrency problem can be reformulated using nodes instead of threads.

Concurrency problem using nodes

1. **a1** on **node1** must occur before **b2** on **node2**
2. **b1** on **node2** must occur before **a2** on **node1**

The **BRPOP** command blocks until a list is non-empty, then removes the last element. The **LPUSH** command pushes an element onto the beginning of the list, which will cause one of the connections blocking on the list to become unblocked. If multiple connections are waiting on the same list, they are served in first come, first serve order [3]. This has the added benefit of having the behaviour of a *strong semaphore*[8]. There isn't any shared code between the nodes this time, but it does require that both nodes have access to the same Redis server or cluster.

Listing 4.9: Code for node1

```
1 a1
2 LPUSH a1Done signal
3 BRPOP b1Done
4 a2
```

Listing 4.10: Code for node2

```
1 b1
2 LPUSH b1Done signal
3 BRPOP a1Done
4 b2
```

The RedisDC library encapsulates this basic principle. RedisDC is written in C and can be downloaded and used for free[9].

Listing 4.11: Obtaining RedisDC

```
1 $ git clone https://github.com/paulmoore/RedisDC
2 $ git submodule update --init
3 $ make
```

Here is the solution to the rendezvous problem using the library:

Listing 4.12: Code for node1 using RedisDC

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "redisdc.h"
4
```

[8]A weak semaphore unblocks threads in random order as opposed to FIFI, and therefore might lead to starvation.

[9]Download RedisDC on GitHub: https://github.com/paulmoore/RedisDC

```
5 // running on node1
6 int main(int argc, char **argv) {
7   char *host = argv[0];
8   int port = atoi(argv[1]);
9   rdc_sem_t *a1Done = rdc_sem_init("a1Done", 0, host, port);
10  rdc_sem_t *b1Done = rdc_sem_init("b1Done", 0, host, port);
11  a1();
12  printf("a1 done\n");
13  rdc_sem_signal(a1Done);
14  rdc_sem_wait(b1Done);
15  printf("a1 and b1 done\n");
16  a2();
17  return 0;
18 }
19
20 // ...
```

Listing 4.13: Code for node2 using RedisDC

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "redisdc.h"
4
5 // running on node2
6 int main(int argc, char **argv) {
7   char *host = argv[0];
8   int port = atoi(argv[1]);
9   rdc_sem_t *a1Done = rdc_sem_init("a1Done", 0, host, port);
10  rdc_sem_t *b1Done = rdc_sem_init("b1Done", 0, host, port);
11  b1();
12  printf("b1 done\n");
13  rdc_sem_signal(b1Done);
14  rdc_sem_wait(a1Done);
15  printf("a1 and b1 done\n");
16  b2();
17  return 0;
18 }
19
20 // ...
```

The above example illustrates the two main semaphore functions, *wait* and *signal*. A semaphore is created with the `rdc_sem_init` function which takes a unique name for the semaphore, an initial value, and a host and port to the Redis Database. In addition to the standard wait and signal functionality, there are some added utility functions.

The function:
`rdc_sem_signal_n(rdc_sem_t *rdc_sem, unsigned int n)`
will signal the semaphore n times. Unlike the strict definition of a semaphore, this operation *is* atomic (and more efficient due to pipelining). A non-atomic version may be implemented in the future.

The function:
`rdc_sem_wait_timeout(rdc_sem_t *rdc_sem, unsigned int timeout)`
will set a timeout in seconds on the blocking operation. If the operation timed out before the semaphore signalled this connection, the function will return 0, and > 0 in all other cases.

RedisDC is not used in the current release of the game, but was built as a proof of concept based on a potential need for it in the future. Bindings to Node.js have yet to be created. Currently, only semaphores are supported, but it is feasible to add other control structures such the mutex, barrier, turnstile, and light switch at a later time.

4.2.2 front-end Stack

The front-end is the application that the user interacts with on their device. It uses a game engine to handle the display logic, and several custom modules to connect to the back-end. Several platforms are available for mobile application development. In the end, I decided to create a native iOS application based on budget, device availability and previous experience.

4.2.2.1 Cocos2d

Cocos2d[10] is a game engine originally written in Python. Its popularity caused it to eventually be ported to Objective-C to target iOS. Since then, numerous improvements have been made. At its core, cocos2d is a game engine which wraps OpenGL. In addition, it provides an object-oriented library to manage scenes, sprites, particle effects, action sequences, and a variety of other game related features. It also interfaces cleanly with physics engines such as box2d[11].

[10] Cocos2d iPhone port website: `http://www.cocos2d-iphone.org/`

[11] Box2d is a well known physics engine designed for games: `http://box2d.org/`

4.2.2.2 CocosBuilder

CocosBuilder[12] is an application designed as a GUI editor for cocos2d based projects. It enables developers to create scenes and animations visually as opposed to writing code. Code connections will enable you to assign items to instance variables and even wire up callbacks for certain events. Version 3.0 and higher is capable of generating sprite sheets from individual assets, and exporting them to different resolutions for different devices and platforms.

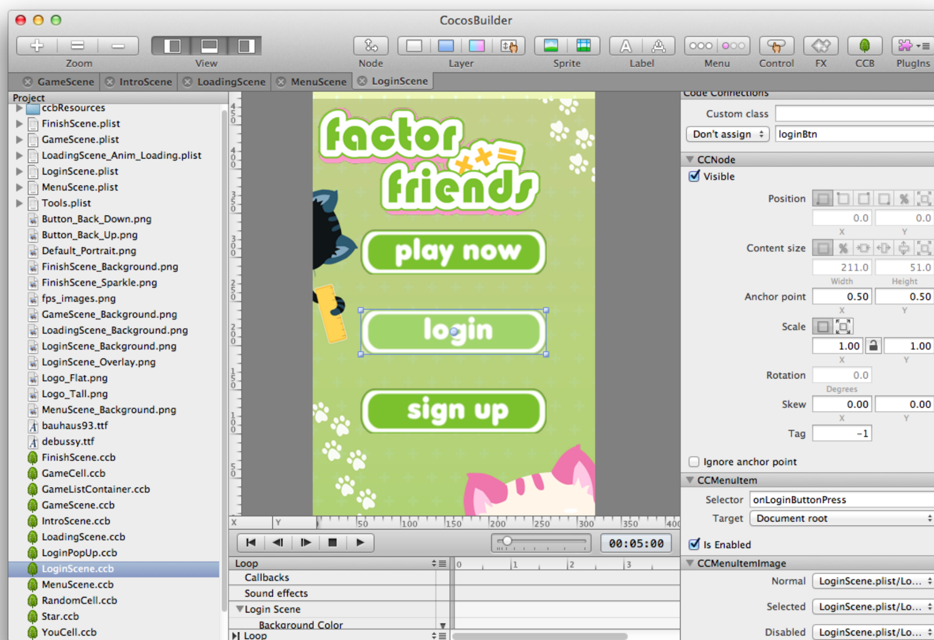

Figure 4.7: CocosBuilder streamlines game development even further with a visual editor.

[12]CocosBuilder website: `http://cocosbuilder.com/`

4.3 REST API

The RESTful web API is based largely on JSON. This section will go over the basic protocol and how client's are authenticated.

4.3.1 Authentication

The Factor Friend's API Server uses a token based authentication scheme. When a user performs a login, a new, randomly generated 16-byte token is generated and sent to the user. For all subsequent requests, the token is not sent in either direction and becomes a shared secret between the client and server.

The client makes an authenticated request by taking the full request body plus timestamp and generating an HMAC using the authentication token as the key. This HMAC is then sent as part of the route, along with the request body, to the server.

The server can then validate the request by recomputing the HMAC with the authentication token. If the HMAC matches the one sent by the client, the user is authenticated.

Listing 4.14: Server-side impementation of the authentication scheme

```
1 if not player.pid @params.pid
2   @log.warn 'Bad request, invalid player ID'
3   next new InvalidArgumentError 'Missing player ID'
4 else
5   @redisConn (conn) =>
6     conn.hget "player:#{@params.pid}", 'token', (err, token)
        =>
7       # The token maybe null if the user was deleted.
8       if err?
9         @log.error err
10        next new InternalError err
11      else if not token?
12        @log.warn 'Player is not logged in - no token'
13        next new NotAuthorizedError 'You are not logged in'
14      else
15        # The HMAC will use the player's token as the key.
16        cipher = createHmac hmac.algo, token
17        # Add a shared salt to the data as well.
18        # This is insecure without also using the token as the
              key,
19        # but adds more unpredictability.
```

```
20        cipher.update @body + hmac.salt
21        hash = cipher.digest 'hex'
22        if hash is @params.hmac
23          # Session validated, both keys are the same.
24          next null, conn
25        else
26          # The keys don't match, either the message
27          # was tampered with or the session is invalid.
28          @log.warn 'Invalid session or HMAC'
29          next new NotAuthorizedError "I'm afraid I can't let
              you do that, Dave"
```

This has two benefits. First, by signing the entire request body with the authentication token, the server is ensured that an attacker did not modify its contents en route. Secondly, if an attacker manages to retrieve one of the clients messages, it still cannot do anything with it. Even if the attacker found a hash collision that produces the same HMAC value for that one request, it is not guaranteed to work with any subsequent requests, as the actual authentication token will produce a different hash value. Because of the timestamp, an attacker cannot even echo the original request again, as the timestamp will be invalidated by the server.

4.3.2 Request and Response Structure

The basic format for any request is as follows:

Listing 4.15: Example HTTP request made to the Factor Friends API web service

```
1 POST https://api.factorfriends.com/:hmac
2 Content-Type: application/json
3 Accept: application/json
4 Accept-Version: 1.0.x
5 {
6   type: ":requestType",
7   pid: ":playerID",
8   ...
9 }
```

Each request POSTs to the API Server with the computed HMAC as the end of the route URL. The client sends the appropriate headers, including the requested API version. Finally, a JSON body is sent along with the request.

Each request requires a *type*. This key tells the server what type of request is being made. Authenticated requests (ones that require the user to be logged in), also require a *pid* (player ID). The current version of the API (1.0.x), supports the following types:

API Request Types

- **login** - Logs the user in as an anonymous user or through Facebook, starting a new active session.
- **logout** - Logs the user out, closing the active session.
- **check** - Check the validity of a session.
- **listen** - Report back any new messages this user should care about.
- **games** - Gather game information for all active games the user is participating in.
- **data** - Gather game information on a specific game the user is participating in.
- **random** - Enter the random game queue.
- **qrandom** - Quit the random game queue.
- **move** - On your turn, make a move in a particular game.
- **device** - Update a user's device credentials so the server can send push notifications to the device.
- **craft** - Validate the user has found a new crafting recipe.

Based on the type, the server initializes an appropriate Request Handler, and immediately allows it to handle the request. This may seem unconventional to a normal RESTful web service, which makes use of URL routes to discern data and HTTP verbs to indicate actions. It is. By placing all of the request information into the request body, and not into the URL, we are able to fully authenticate it. This separates the authenticated data, the JSON request body, from the signature, the unique URL. Otherwise, the route and any URL parameters would need to be authenticated separately, and then somehow recombined. And so, it is a cleaner design to take the HMAC of the request body keyed to the authentication token, at the expense of strict REST design.

Listing 4.16: The server selects an appropriate Request Handler based on the type field

```
1 # Each API request is a POST to the same URL with an HMAC and
     a JSON body.
2 app.post
3   path: "#{server.path}/:hmac"
4   name: api.name
5   version: api.version,
6   (req, res, next) ->
7     # Only handle JSON requests.
8     if req.is('application/json') and req.accepts('application
         /json')
9       req.log.debug request: req.params, 'REQUEST'
10      # Find the appropriate message handler.
11      handlerType = handlers[req.params.type]
12      if handlerType?
13        req.log.debug "Handling #{req.params.type} with #{
            handlerType}"
14        klass = require handlerType
15        handler = new klass req.log, req.body, req.params, res
16        handler.handleRequest()
17      else
18        req.log.error "No request handler for #{req.params.
            type}"
19        res.send new InvalidArgumentError
20          "Unrecognized handler: #{req.params.type}"
21    else
22      req.log.error "Sent invalid request: #{req.body}"
23      res.send new WrongAcceptError
24        'Content-Type and Accepts headers must be application/
            json'
25    next()
```

4.4 Client-side Networking

4.4.1 Airtower

The client-side communication module to contact the web service is named **Airtower**. The Airtower module has classes which encapsulate the underlying REST protocol.

Airtower Module

- **AirtowerRequest** *(class)* - An instance of AirtowerRequest represents one, single-use request that can be made to the server. It is capable of serializing and sending the request to the server and handling the response. Subclasses of AirtowerRequest are used for each specific request.
- **AirtowerResponse** *(protocol)* - An instance of a class conforming to AirtowerResponse represents a response from an AirtowerRequest. It is capable of deserializing a response from the server. Each type of response has a custom class which conforms to this protocol.

Listing 4.17: Example AirtowerRequest implementation

```
1 @interface FacebookLoginRequest : AirtowerRequest
2 @property (nonatomic) FBSession *session;
3 - (id)initWithSession:(FBSession *)session;
4 @end
5
6
7 @implementation FacebookLoginRequest
8 @synthesize session = _session;
9 - (id)initWithSession:(FBSession *)session
10 {
11     if ((self = [super init])) {
12         self.session = session;
13     }
14     return self;
15 }
16 - (NSDictionary *)JSONRequest
17 {
18     return @{
19         @"type": @"login",
20         @"method": @"fb",
21         @"fbaccess": self.session.accessToken
22     };
23 }
24 @end
```

Listing 4.18: Example AirtowerResponse implementation

```
1 @interface LoginResponse : NSObject <AirtowerResponse>
2 @property (nonatomic, copy) NSString *pid, *token, *method, *
    nickname;
3 @property (nonatomic) NSUInteger msgc;
```

```
4 @end
5
6 @implementation LoginResponse
7 @synthesize pid = _pid;
8 @synthesize token = _token;
9 @synthesize method = _method;
10 @synthesize msgc = _msgc;
11 @synthesize nickname = _nickname;
12 - (id)initWithInfo:(id)info
13 {
14     if ((self = [super init])) {
15         self.pid = [info objectForKey:@"pid"];
16         self.token = [info objectForKey:@"token"];
17         self.method = [info objectForKey:@"method"];
18         self.msgc = [[info objectForKey:@"msgc"]
              unsignedIntegerValue];
19         self.nickname = [info objectForKey:@"nickname"];
20     }
21     return self;
22 }
23 @end
```

Using the Airtower A request can be made to the web service by creating a new instance of an AirtowerRequest. Then, optionally, callbacks can be added to the request object to handle any response. The callbacks take the form of closures, implemented using C Blocks.

Listing 4.19: Making a request to the web service using the Airtower

```
1 AirtowerRequest *req = [[FacebookLoginRequest alloc]
      initWithSession:[FBSession activeSession]];
2 req.callback = ^(FacebookLoginRequest *request, LoginResponse
      *response) {
3     [Login beginActiveLoginWithPid:response.pid
4                              token:response.token
5                             method:response.method
6                               date:[NSDate date]
7                               msgc:response.msgc
8                           nickname:response.nickname];
9     FFLog(@"Login success: %@", [Login activeLogin]);
10 };
11 req.errorback = ^(FacebookLoginRequest *request, ErrorResponse
       *response) {
12     [session closeAndClearTokenInformation];
13     FFLog(@"Login failed: %@", response);
```

```
14 };
15 [req start];
```

4.4.2 Secure Socket Layer

To further increase the security of client and server communication, HTTPS is used instead of unencrypted HTTP. Normally, this involves submitting a request for a certificate from a trusted Certificate Authority. However, for development purposes, a self-signed certificate is good enough for testing purposes. This presents a problem, because iOS does not by default allow an HTTPS connection when the host is using a self-signed certificate. We can get around this by adding the following methods to `AirtowerRequest`, which implements the `NSURLConnectionDelegate` protocol. This overrides the default behaviour to deny all self-signed certificates when we are in `DEBUG` mode.

Listing 4.20: Allowing self-signed SSL certificates originating from factor-friends.com in DEBUG mode

```
1 #ifdef DEBUG
2 - (BOOL)connection:(NSURLConnection *)connection
     canAuthenticateAgainstProtectionSpace:(
     NSURLProtectionSpace *)protectionSpace
3 {
4     if ([protectionSpace.authenticationMethod isEqualToString:
          NSURLAuthenticationMethodServerTrust]) {
5         return YES;
6     } else {
7         FFLog(@"Can't authenticate with method: %@",
              protectionSpace.authenticationMethod);
8         return NO;
9     }
10 }
11
12 - (void)connection:(NSURLConnection *)connection
     didReceiveAuthenticationChallenge:(
     NSURLAuthenticationChallenge *)challenge
13 {
14   if ([challenge.protectionSpace.authenticationMethod
        isEqualToString:NSURLAuthenticationMethodServerTrust]) {
15     if ([challenge.protectionSpace.host isEqualToString:[[
          Config sharedConfig] stringSetting:FFConfigServerHost
          ]]) {
```

```
16        NSURLCredential *credential = [NSURLCredential
              credentialForTrust:challenge.protectionSpace.
              serverTrust];
17        [challenge.sender useCredential:credential
              forAuthenticationChallenge:challenge];
18      } else {
19              FFLog(@"Not going to trust: %@", challenge.
                  protectionSpace.host);
20          }
21    } else {
22          FFLog(@"Wrong authentication method: %@", challenge.
              protectionSpace.authenticationMethod);
23      }
24    [challenge.sender
        continueWithoutCredentialForAuthenticationChallenge:
        challenge];
25  }
26  #endif
```

4.5 Implementing Game Play

While the gameplay of Factor Friends is fairly simple, it presented itself with some difficulties in its implementation.

4.5.1 Evaluating Equations

A major component of the game is evaluating mathematical equations. A simple parser is capable of basic algebraic operators, it becomes much more complicated once functions and unary operators are thrown into the mix. In addition, the library has to be compatible with both the server and client, considering both systems will need to compute these equations.

4.5.1.1 The bet.coffee Infix Equation Evaluator for CoffeeScript

This led to the development of a submodule to the API Server named **bet.coffee**[13], originally after Binary Expression Trees. the bet.coffee module has been open sourced and can be used with any Node.js or Browser based application.

[13] Download the bet.coffee package on GitHub: `https://github.com/paulmoore/BET`

Installing To install the library, include the source files in your project or use NPM:

```
$ npm install bet
```

4.5.1.2 API

The library expects that you have some way of separating your tokens, whether it is a `string.split`, or something more involved, is up to you. For instance, if we wanted to evaluate $1 + \lfloor\sqrt{2^2}\rfloor$ we would define an equation as follows:

Listing 4.21: Input to the bet.coffee library

```
1 eqn = ['1','+','isqrt','(','2','^2',')']
```

It can then be evaluated using the *evaluate* function.

Listing 4.22: Evaluating an infix equation asynchronously using the bet.coffee library

```
1 # Require the module
2 {evaluate} = require 'bet'
3 # Evaluating an equation
4 evaluate eqn, (error, result) ->
5     console.log error ? result
```

The library also provides a synchronous function *evaluateSync*

Listing 4.23: Synchronous API of the bet.coffee library

```
1 {evaluateSync} = require 'bet'
2 try
3     val = evaluateSync [1, '+', 2]
4     console.log val
5     # throws an error, invalid equation
6     evaluateSync ['*', '+', 1, 'x']
7 catch e
8     console.log e
```

4.5.1.3 The Shunting-Yard Algorithm

The library takes equations in *infix* notation. This means the operators are placed between the operands. The problem with evaluating infix equations is that order of operations must first be determined and applied to the equation incrementally.

Reverse Polish Notation RPN notation places the operators *at the end* of the operators. The equation is ordered such that the order of operations is intrinsic to the equation. For instance, taking the infix equation `1 + 2 * 3 + 4 ^ 5`, the same equation in RPN format is `1 2 3 * + 2 3 2 ^ ^ +`. The advantage to this format is that it can now be evaluated directly.

Shunting-Yard Algorithm The Shunting-Yard algorithm takes an infix equation and produces its equivalent RPN format. This is the algorithm that the bet.coffee library uses to evaluate arithmetic equations.

Listing 4.24: Pseudocode for the Shunting-Yard Algorithm

```
1 tokens := equation in infix notation
2 queue := an empty queue
3 stack := an empty stack
4 while tokens is not empty:
5   token = tokens.pop()
6   if token is Number:
7     queue.enqueue(token)
8   else if token is Function:
9     stack.push token
10  else if token is Comma,
11    until stack.peek() is Left Parenthesis:
12      output.enqueue(stack.pop())
13  else if token is Operator:
14    o1 := token
15    if o1.fix = 'pre':
16      stack.push(o1)
17    else if o1.fix = 'post':
18      output.enqueue(o1)
19    else
20      while stack is not empty:
21        if stack.peek() is Operator:
22          o2 := stack.peek()
23          if o1.assoc = 'left' and o1.prec <= o2.prec or o1.
              prec < o2.prec
24            output.enqueue(stack.pop())
25            continue
26        break
27      stack.push(o1)
28  else if token is Left Parenthesis:
29    stack.push(token)
30  else if token is Right Parenthesis:
31    until stack.peek() is Left Parenthesis:
```

```
32        output.enqueue(stack.pop())
33      stack.pop()
34 while stack is not empty:
35   queue.enqueue(stack.pop())
36 return queue
```

The output of the algorithm produces a queue which represents the equation in RPN notation. Using a stack, the equation can then be evaluated using the following procedure:

Listing 4.25: Evaluating equations in Reverse Polish Notation

```
1 queue := output from the Shunting-Yard Algorithm
2 stack := new empty stack
3 result := Not a Number
4 while stack or queue is not empty
5     if queue is not empty
6         token := queue.dequeue()
7         stack.push(token)
8     if stack is not empty
9         fnop := stack.peek()
10        if fnop is Operator or Function and stack.length >
              fnop.argc
11            stack.pop()
12            args = new array of size fnop.argc
13            for i := fnop.argc to 1
14                args[i - 1] := stack.pop()
15            result = fnop.exec args
16            if stack or queue is not empty
17                stack.push(result)
18 return result
```

Evaluating an equation in Reverse Polish Notation notation

1. The procedure loops while the queue or the stack are not empty.
2. If the queue is not empty, a token is dequeued from it and pushed onto the stack.
3. If the token at the top of the stack is a function or operator, and there are enough operands on the stack to support that operator or function, pop all of them from the stack.
4. Evaluate the operator or function given the operands popped from the stack
5. Repeat

4.5.1.4 Custom Operators

One advantage to the **Shunting-Yard Algorithm** is that it is trivial to implement custom operators and functions. One can add a custom operator by adding a definition to the **operators** object of the **bet** package. Here is an example of adding a C-style logical AND operator:

Listing 4.26: Creating or redefining an operator using bet.coffee

```
1 {evaluate, operators} = require 'bet'
2 operators['&&'] =
3     assoc: 'left'
4     prec: 0
5     argc: 2
6     fix: 'in'
7     exec: (args) -> 1 if args[0] isnt 0 and args[1] isnt 0
         else 0
8 evaluate [1,'&&',1,'&&',0], (error, result) ->
9   console.log error ? result
```

Operator Attributes

- **assoc** - Associativity *['left' or 'right']* Associativity indicates the order in which operators of the same precedence are executed. For instance, && has an associativity of 'left' and thus a && b && c is evaluated as (a && b) && c.

- **prec** - Precedence *[integer]* Operators with a higher precedence (higher value) are executed first. For instance, $1+2*3$ is evaluated as $1+(2*3)$.

- **argc** - Argument count *[integer]* The number of numerical operands an operator needs to execute. In practice this is usually only 1 (for unary) or 2 (for binary) operators. For instance, + requires 2 operands e.g. $1+2$, whereas $1+$ will produce an error.

- **fix** - How the operator is 'fixed' *['in', 'pre', or 'post']* Most binary operators are infixed, meaning the operator is between the operands e.g. 1/2. Unary operators are usually either pre or post fixed, e.g. 5! (postfixed) or $not1$ (prefixed). However, you can also have infixed unary operators (just be careful with associativity!) such as pre and post increment/decrement, e.g. $++1$ and $1++$ are both valid.

- **exec** - Evaluator *[function]* This is the function that is called to evaluate the operator. It is given a single argument as an array, with length *argc*. All values are numerical.

4.5.1.5 Custom Functions

Functions are similar to operators. You can also define new or redefine functions. Functions in this library are invoked C style $fn(arg1, arg2, arg3)$. Currently, variable argument functions are not supported. Function arguments can be expressions in themselves. Functions cannot have the same name as an operator.

Listing 4.27: Creating or redefining a function using bet.coffee

```
1 {evaluate, functions} = require 'bet'
2 # Averages 3 numbers
3 functions['avg'] =
4     argc: 3
5     exec: (args) -> (args[0] + args[1] + args[2]) / 3
6 evaluate ['avg','(',1,2,3,')'], (error, result) ->
7     console.log error ? result
```

Declaration of a function is much like an operator. However it requires only two attributes to be defined.

Function Attributes

- **argc** - Argument count *[integer]* The number or arguments the function takes.

- **exec** - Evaluator *[function]* - This is the function that is called to evaluate the function. It is passed an array of in order numerical arguments.

4.5.1.6 Crossplatform Implementation

Single code base Since the iOS API allows arbitrary JavaScript code to be executed, as we will see later, this allowed for the development of a single implementation of the library. Otherwise, a CoffeeScript and Objective-C version would have had to been co-developed.

The bet.coffee package was written to not require any Node.js dependencies. However, it can be loaded as a module into any Node or Browser project. This is a required feature as the client does not have the Node.js libraries available to it. In this case, the module functions are placed in the *window.BET* object, which we will see in the next section.

4.5.1.7 Client Wrapper

UIWebView To run bet.coffee on iOS, we first need a method for executing JavaScript. Luckily, a standard `UIWebView` will do the trick. A web view is normally meant for displaying a web page in an app. This means that it also includes a JavaScript environment. Factor Friends uses an invisible web view without a loaded page, and uses it to run bet.coffee.

Setting up the code library First, we have to load the library into the web view's JavaScript environment. This means loading the library from the applications bundle (stored on disk). A `UIWebView` has a single method to interface with the JavaScript engine, `stringByEvaluatingJavaScriptFromString:`. This method takes a JavaScript string and evaluates it as an expression, returning the result to the caller.

Listing 4.28: Setting up bet.coffee in a web view

```
1 - (void)setupCodeLibrary
2 {
3     NSError *error = nil;
4     NSString *js = [NSString stringWithContentsOfFile:[[
          NSBundle mainBundle] pathForResource:@"BET" ofType:@"
          js"] encoding:NSUTF8StringEncoding error:&error];
5     if (error) {
6         FFLog(@"Error loading BET.js! %@", [error description
              ]);
7     } else {
8         [self.webView stringByEvaluatingJavaScriptFromString:
              js];
9     }
10 }
```

Interfacing with bet.coffee Next, we need a way of invoking the library functions from our code. To do this, a small piece of code is used to call the library function with the proper arguments. We use the JavaScript self-invoking function module pattern to avoid any variable leaking. Notice the `%@` formatter at the end of the string. This will allow us to set the function's argument to whatever we specify later.

Listing 4.29: Interfacing to bet.coffee

```
1 static NSString* const JSEvaluate =
2 @"(function(eqn) {"
```

```
3 @"  var ret = NaN;"
4 @"  BET.evaluate(eqn, function(error, result) {"
5 @"      ret = result;"
6 @"  });"
7 @"  return ret;"
8 @"}).call(this, %@);";
```

Calling the library Finally, we can actually use the library. To do this, we need the input to the `evaluate` function, which is a JavaScript array encoded as a string. This can be done any number of ways (using a JSON serializer for example), but won't be shown here. The method shown in the following listing:
`- (BOOL)evaluate:(NSString *)eqn resultRef:(NSInteger *)presult`
expects an equation string, and a pointer to store the result to. If there was an error with the library or the equation was invalid, the method returns NO. Otherwise, it returns YES and stores the equation's result in the result pointer.

Listing 4.30: Evaluating an equation from Objective-C

```
1 - (BOOL)evaluate:(NSString *)eqn resultRef:(NSInteger *)
    presult
2 {
3     NSString *result = [self.webView
          stringByEvaluatingJavaScriptFromString:[NSString
          stringWithFormat:JSEvaluate, eqn]];
4     if (!result || result.length == 0 || [result
          isEqualToString:@"NaN"]) {
5         // invalid equation
6         return NO;
7     }
8     if (presult != NULL) {
9         NSInteger value = [result integerValue];
10        *presult = value;
11    }
12    return YES;
13 }
```

4.5.2 Game Messages

Game Messages are important events A standard request receives one response from the server. Game messages differ from a standard request in

that they propagate somewhere else and are delivered to the client in an asynchronous fashion. In other circumstances, they would be called *events*, as they do not have a 1 : 1 relationship like the request/response system is structured. For instance, if a player makes a move on his or her turn, we need to notify the opponent that the game has changed state. In addition, a game message is considered vital for the client to operate properly. This means each game message *cannot* be lost or remain unsent indefinitely, and must be received in the correct order that they were created. This posed some problems in that Redis does not offer full transactional support and the communication model does not maintain an open channel between the client and the server.

Types of Messages Game messages, like requests, have a type to identify how the message should be handled. In addition, just like requests, additional information about the event is sent in JSON format. Below is a list of currently available message types that the server may produce.

Game Message Types

- **gj** - Game Join: The player has joined a new game.
- **gm** - Game Move: An opponent has made a move in one of the player's active games.
- **gq** - Game Quit: An opponent has quit or was removed from the game, the game is now over.
- **gf** - Game Finish: The last move was made in a game and is now finished, results from the game should be collected.
- **fl** - Forced Logout: The server has logged the user out of the active session. Most likely caused by logging in to the account from another device.
- **su** - Service Unavailable: The server is going to be restarted or temporarily unavailable.

QueueListen An algorithm called QueueListen (QL) was developed to guarantee delivery of asynchronous events. QL works by assigning a message ID to each game message. Messages get added to a queue for each player

when they are created. The client sends the last known ID to the server when it requests to check the message queue. Messages are only deleted once the server is assured the client has received them based on the client's message ID. A message is delivered in two phases: *queueing* the message for delivery, and receiving the message by having the client *listen* for it.

4.5.2.1 Sending a Message

Queue Phase The first stage to delivering a message is to add it to the player's message queue. The message queue is analogous deposit box where they are queued until the client is ready for them to be received. The following script is the query used to send a message. It can send multiple messages to multiple recipients with one call.

Listing 4.31: The Lua script to add an event to a player's message queue

```lua
-- KEYS contains the player ids to send to messages to.
-- ARGV contains the messages to send.
for _, pid in ipairs(KEYS) do
  -- Need access to the player, message queue, and player
     channel keys.
  local playerKey = "player:"..pid
  local queueKey = "msg_queue:"..pid
  local channelKey = "msg_channel:"..pid
  -- For each message, increment message index.
  local msgIndex = redis.call("hincrby", playerKey, "msgc", #
     ARGV)
  for k, msg in ipairs(ARGV) do
    -- Add the message to the end of the message queue.
    redis.call("zadd", queueKey, msgIndex, msg)
  end
  redis.call("publish", channelKey, "q")
end
return true
```

How it works First, the player's message counter is incremented. This new value will be used as the unique ID for the message (known as a message index). Then, the message is added to a sorted set keyed to the message index. Finally, a message is published to the player's channel. The contents of this message are not important, only in that they signify to any receiving nodes that a new message has been saved.

4.5.2.2 Receiving a Message

Listen Phase The second stage is to wait for the client to come online and ask for an updated list of messages. This is done by sending a **listen** request to the API Server. The server will then either return right away with new messages, or block until either it times out or a new message has been generated.

Listing 4.32: Pseudocode for receiving new messages with a **listen** request

```
1 current_index := the latest message index in the player's
      message queue
2 last_known_index := the client's last known message index
3
4 if last_known_index < current_index:
5   finish()
6 else
7   subscribe to the player's channel
8   block until a message is received or the request times out
9   finish()
10
11 function finish():
12   remove all messages from the queue with index <
        last_known_index
13   messages := get all remaining messages in the queue
14   current_index := get the latest message index from the queue
15   send_response(messages, current_index)
```

Check This procedure works by first checking if there are any new messages for the player by comparing the current message index to the one the client sends in the request. If the current message index is greater than that of the clients, the procedure moves into the finish phase.

Wait Otherwise, the server subscribes to the player's pub/sub channel[14]. The server will unblock if another node sends a *"message received"* event. It will also unblock if it times out after a certain period.

Finish Finally, the server will perform the last operations on the queue. First, all of the messages that are older than specified by the client's last

[14] Each player is given a unique channel name. Redis supports publish and subscribe operations that may block until a new message is received. More information here: `http://redis.io/topics/pubsub`

known message index are deleted. All of the remaining messages are extracted from the queue, but not deleted. Lastly, these messages, along with the latex message index are sent to the client.

4.5.2.3 Long Polling

Real-time updates Because the server may not respond right away, this type of request is known as *long polling* or a *comet request.* This method has the advantage that the client does not need to repeatedly poll the server to get near real-time updates. The server will instead hold the connection open until an event occurs, to which it can then send a response.

4.5.2.4 Correctness

Proof of correctness Each message is not deleted until it has confirmation that it arrived successfully. This confirmation comes from when the client initiates another **listen** request with the updated message index. If the **listen** request fails for whatever reason, the client's last known message index is not updated. If the message index is not updated, the client will resend the old message index during the next request. Because the server does not delete a message until the client sends an index greater than it, the server will attempt to resend the message. Each message is sent in the order in which it was queued. Thus, each message is guaranteed to be delivered in order, and no message is lost due to a failure on the client or server.

Further improvements A further improvement to the algorithm would be to have the Database save a backlog of messages greater than what is required to ensure integrity. This would allow separate devices attached to the same account receive previous messages if necessary.

4.5.2.5 Client-side Game Message Handling

A QL receiver module is implemented in the client to handling incoming game messages. The module works by making the long polling **listen** request repeatedly. It also acts as an event dispatcher. If messages were received, they are dispatched to the appropriate listener.

The below example illustrates how the menu scene listens for different types of game messages. When a message occurs, the receiver will dispatch an event to the scene and the scene will update the game list accordingly.

Listing 4.33: The menu scene listining for game messages

```
1 QueueListenReceiver *receiver = [QueueListenReceiver
     sharedReceiver];
2 [receiver forConfigValue:FFConfigMessageGameJoin setListener
     :^(GameJoinMessage *msg) {
3     if (msg.src == RandomSource) {
4         [list setRandomCellVisible:NO];
5     }
6     [list addGame:msg.game];
7 }];
8 [receiver forConfigValue:FFConfigMessageGameMove setListener
     :^(GameMoveMessage *msg) {
9     [list changeGameStatus:msg.game];
10 }];
11 [receiver forConfigValue:FFConfigMessageGameQuit setListener
     :^(GameQuitMessage *msg) {
12     [list removeGame:msg.game];
13 }];
14 [receiver forConfigValue:FFConfigMessageGameFinished
     setListener:^(GameFinishedMessage *msg) {
15     [list finishedGame:msg.game];
16 }];
```

4.6 Additional Resources

In addition to what has been discussed in this article, more technology was developed for Factor Friends which was not discussed here. There are several other useful libraries which I have open sourced during development listed below:

- Loading Facebook profile pictures into cocos2d: `http://paul-moore.ca/blog/2013/01/28/facebook-profile-pictures-in-cocos2d/`
- Cocos2d utility library: `https://github.com/paulmoore/CocosUtils`
- The original Factor Friends prototype: `https://github.com/paulmoore/Factor-Friends`

Chapter 5

Conclusions

Technology is enabling learning in drastically new ways. If mobile devices are to be taken more seriously in education, they need to be utilized to their maximum potential. More effort needs to go into integrating features that have worked incredibly well for other applications. Social integration, push-based content, multiplayer, and fun theory are all concepts that have made a very positive impact in the mobile market.

Video games are just one more tool to deliver something of value to a user. In this case, we want to deliver rich educational software on handheld devices that people are motivated to use. If designed properly, video games are incredible tools that can promote new ways of learning.

This is what Factor Friends does. It combines the proven concepts of a successful mobile application with game design techniques that works. To make math more interesting, Factor Friends uses the game like properties of math to make math the primary component of the game, instead of an obstacle around the fun.

In addition, Factor Friends uses scalable and extensible technology so that future titles can be built on top of the same platform.

The problem isn't that subjects such as math and computer science are not being taught correctly. There are many great resources for learning a science. The problem is these topics need to be made more fun. If students find these topics interesting, they will be motivated to learn more.

If you want to learn more about this project, please visit Factor Friend's website at `http://factorfriends.com` or email me at `info@factorfriends.com`.

Bibliography

[1] *American Families See Tablets as Playmate, Teacher and Babysitter Comments Feed.*, Nielsen. Available at: http://www.nielsen.com/us/en/newswire/2012/american-families-see-tablets-as-playmate-teacher-and-babysitter.html

[2] Ryan Dahl *History of Node.js*, YouTube. Available at: http://www.youtube.com/watch?v=SAc0vQCC6UQ

[3] Salvatore Sanfilippo *BLPOP Documentation*, Redis. Available at: http://redis.io/commands/blpop