PokeAdemics

"Translate academic success into cash"

Bradley Bernard, bmbernar@ucsc.edu
Aidan Gadberry, agadberr@ucsc.edu

March 12, 2016

Motivation

As computer science students we are always trying to create the next viral application, the next Facebook, Snapchat, or Uber in between studying for our classes. The idea behind our project sprung from the idea to attempt to create an authentic application that students at our university could use and is fully functional to be able to release into the an app store.

Our idea, PokeAdemics, was born through competition on grades with our classmates. Most of our friends in computer science are very competitive and we wanted to reward them and promote competition through academic performance. Now students can put their money where their mouth is, and all claim their rightful cash from their friends!

We decided on an iPhone app because we both had very little experience in coding in Swift or deploying iOS applications. We chose PHP (v7.0) for the backend because it has great frameworks and tons of community packages that we needed to lift our idea off the ground. We could have chosen Objective-C for the iPhone app, but we wanted to be bleeding edge and take up the new Apple programming language Swift with its latest version (v2.2). We chose PHP over other back end languages because we were told of a very powerful PHP framework that allowed for rapid application development called Laravel (v5.2).

Project Overview

Our project is essentially two parts: an iPhone client in Swift and a RESTful (representational state transfer) API (application programming interface) in PHP. The iPhone client would send HTTP requests to our PHP code, which would then return JSON and the iPhone would decide what to do with the JSON to display it. Overall, our project was difficult; a lot goes into making the communication smooth and predictable between client and server.

Authentication

Since there are two separate parts, we initially needed to decide on an authentication protocol for our server to be able to identify who was requesting what resource from our API. We looked up new authentication protocols and landed using **JWT**s (JSON Web Token).

A JWT (https://jwt.io/) is JSON information Base64 encoded and then signed by a secret key. It is very lightweight and you can verify it came from the server by checking if the data is encrypted correctly using a the same key client side. An example JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95
OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

The information we included in our signed JWT was just the User ID of the authenticated user. So when the iPhone client sent us a request, it included the JWT it got from the server previously as an HTTP header in the request:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIs…
```

This allowed us to decode the token server side, and see which user was making the request.

Scraping UCSC Servers

Our project relied heavily on the https://my.ucsc.edu/ servers' data so we had to figure out a way to take that data and store it ourselves. There a few things we needed to perform:

- Check if (Cruz ID, Gold Password) combos were valid
- Once logged in, access grade data for a specific quarter
- Once logged in, access classes enrolled for a specific quarter
- Take all classes from the class search system

This was one of our biggest hurdles in our project. The UCSC servers are very fragile; about half of the time our valid requests (logging in, changing selected term) would return errors.

To combat this inconsistency, we implemented our scraping code to use a **do-while** loop in PHP. There was a difference between *invalid* responses and *error* responses. Errors were messages like invalid login details, but invalid responses were errors even though the data was correct. We basically used a do-while loop to try ten times to send data to the server, if we received an *error* response we told the client invalid input. If we received an *invalid* response we incremented our counter and retried our request after sleeping one second so we didn't overload the servers.

Also, the MyUCSC portal that us students login from contains an iframe (another web page), which actually has the data we needed to scrape. So instead of using URLs like my.ucsc.edu we used a different domain: ais-cs.ucsc.edu to get the data like grades, classes, etc. We did not anticipate the trouble it would be getting the data correctly, so we spent a majority of the initial time of the project on the scraping. The Google Chrome developer tools helped us a lot in identifying the network requests (AJAX) that the page made and replicating them in CURL.

Grading Algorithm

Another hurdle we didn't anticipate was the complexity of the grading system and how it worked with ties within a database. When logging in as a student you receive grades for multiple classes, and if any of those grades are entered (professor put them out) we wanted to kickoff a sub routine to fill in grades for all the other users entered in that specific class for all bets. We thought it sounded easy, but in reality it was extremely tough.

CheckGrades

1. We mark one user per bet group as the designated grade checker, as in we will login to this user's account and check their grades every X minutes in by utilizing Cron. Cron jobs run a Linux command at a regular interval. We rotate through a designated user every minute and check their grades, by calling our PHP script every minute.

UpdateGrades

2. Our Cron job calls our PHP code, which selects a user in a bet group and checks his or her grades. Once we login, we see all of their grades for all of their classes. We insert all the grades for that user IF they are in a bet with that class. Then we push a **Job** onto a FIFO queue powered by beanstalkd. The job finds all the other users that have a bet in that specific class and logs in to their portal to insert their grades. At the end of that job, we push another Job onto the queue, to rank the users in each bet group by their grade and notify the winners and payout.

FinishGroup (Part 1 of 2)

3.1 To finish a bet group, we have to rank all the participants of the group by their grades. When we fetch the grade and input it into the database, we select the grade points so we can do a simple sort to see the highest ranked participants. The problem is with ties. There are not a lot of unique combinations of grade points so the likelihood of a tie for a class is extremely common. When users create a bet group, they input the number of participants paid out but what happens if there is a tie?

We have no way to decide which 20.00 grade points (A) is better, if there is a tie. So we had to rethink our paid participants to possibly extend to the number of users in the group. As long as not EVERYONE wins we can payout because if everyone wins, they all get their money back.

So we created an algorithm to assign ranks with ties, meaning if two users are tied their rank is the same and the next distinct participant's rank will not be consecutive from the last participants rank. Example:

User ID	Grade	Credits	Grade Points	Rank
1	A (4.0)	5.0	20.00	1
2	A (4.0)	5.0	20.00	1
3	B (3.0)	5.0	15.00	3 (**not** 2)
4	C (2.0)	5.0	10.00	4

If the above bet group had paid participants set to one (number of users to payout) then it would have gotten extended to two because there was a tie. However if the number of paid participants is extended so much that it is equal to the number of participants in the group (everyone wins) then the bet group ends as a **draw**.

FinishGroup (Part 2 of 2)

3.2 Now that we have our ranks settled for each participant along with ties we now need to pay each winner the right amount. We created another algorithm to do this, one that loops through each winner and inside that, loops through each loser to pay the winner a portion of their winnings.

If we use the example in the previous table, with paid participants set to one and buy in set to $5.00 the winner would win $15.00.

```
winTotal = (buy_in * (participants.count – winners.count))
```

However since there were two winners (a tie between two users), our **winTotal** would actually be $7.50 (winners share the pool). So each winner gets $7.50 from the group of losers, as a whole. So we then loop through the each winner and then loop through each loser and pay the current winner from the outer loop a cut of the **winTotal**, **loserCut.**

```
loserCut = (winTotal / losers.count)
```

In this example, there are two losers and two winners so each winner will get two payments (one from each loser) which will add up to his or her **winTotal**. We also send push notifications to the winner and loser notifying them of their payments and bet results.

```
.pay(destination user, $ amount)
.push(string for push notification to device)

foreach(winners as winner)
  foreach(losers as loser)
      loser.pay(winner, $3.75)
      loser.push("Paid $3.75 to winner.name")
      winner.push("Won $3.75 from loser.name")
```

Database Overview

The core of our application is backed by a robust relational database. We are using MariaDB as our database server. We previously used MySQL but found MariaDB to be faster and less resource intensive for a LEMP stack.

Database Schema

Our database was designed right from the start of our idea to map out how our app would work, and to layout our API from the data side of things. We have ten tables, two of which are used solely by our PHP framework. The eight important ones (created by us) are:

- **groups** (Bet groups)
- **school_classes** (Classes from class search)
- **user_classes** (Relationship table for user to classes)
- **user_grades** (Relationship table for user to grades)
- **user_groups** (Relationship table for user to groups)
- **user_payments** (Log of payments from one user to another)
- **user_stripes** (Contains users' stripe account details and API keys)
- **users** (Main table storing cruz_id, cruz_password, device_token for push notifications…)

To summarize, we have three main tables: **users**, **groups** and **school_classes**. Then we have our two relationship tables, one to tie a user to a group: **user_groups**. Another relationship table to tie a user to class: **user_classes**. Three lookup tables that have a one-to-many relationship with the users table: **user_grades**, **user_stripes**, **user_payments**.

Our database is designed to make it very easy to access data by using one or multiple **JOIN**s in SQL (structured query language). As an example, lets get the users in the Bet Group with ID of 1. So we have multiple users signed up for a **group** and that group has ID of 1.

```
SELECT users.*
FROM users
INNER JOIN user_groups
ON user_groups.user_id = users.id
WHERE user_groups.group_id = 8;
```

This will return all **users** attached to a Bet group with ID of eight. We use this on our bet detail page (when you click on a bet it takes you to another detail view). We show the current users in a bet group, along with their winTotal, place, and other info if the bet is complete (graded, not in progress).

We can do one more, to showcase more joins. We will select the classes for a user of ID 1.

```
SELECT school_classes.*
FROM school_classes
INNER JOIN user_classes
ON user_classes.class_id = school_classes.class_number
INNER JOIN users
ON users.id = user_classes.user_id
WHERE users.id = 1
```

This will return all **school_classes** that a user is currently enrolled in. We use this query on the profile page to show a user's classes.

Project Architecture

Our project's server is hosted through DigitalOcean and is accessible at: https://pokeademics.com. We are using LetsEncrypt to automatically provision an SSL certificate to provide httpS. Our server is running Ubuntu 14.04 LTS with a LEMP stack installed. LEMP stack consists of Linux, Nginx, MariaDB, PHP. We also have beanstalkd installed for our queue manager. We are using Pusher to send information over web sockets to the iPhone client. We are using Apple's Push Notification Service (APNS) to send push notifications to the user when the bet group is completed and graded. To power our payments, we are using Stripe to store users cards securely and send payments using managed accounts on Stripe. We are using JSON Web Tokens for our client/server authentication.

Conclusion

We had a lot of fun and a lot of pain making this app and its back end API. Though we probably spent more time than we should have because we chose such an ambitious project, we are glad we did it because we learned a ton about Swift and PHP. We can showoff a near 100% usable app on our Github and possibly launch it in the future (when we make it more secure). At times, we thought we would come up short, but we managed to make it past through grueling bugs and very tedious algorithms. Also helped us a lot to have a 24-hour extension. Overall we are very happy with what we have created in the short span of a quarter that we had to work on it.

Code Files Reference

For the PHP code it may be hard to find our code for certain things because it is in a framework but we will list the files we created/changed below:

- `app/Console/Commands/CheckGrades.php`
- `app/Console/Commands/FetchAllClasses.php`
- `app/Console/Kernel.php`
- `app/Http/Controllers/APIController.php`
- `app/Http/Controllers/StripeController.php`
- `app/Http/Controllers/UCSCController.php`
- `app/Jobs/FetchUserDetails.php`
- `app/Jobs/FinishGroup.php`
- `app/Jobs/UpdateGrades.php`

All of our database schema in PHP:

- `database/migrations/*.php`

Screenshots and Screencasts:

We realize it might be hard to run our app code so we provided more info. Here are screenshots of our push notifications when people join a group and when results are done from a bet group:

Imgur album link: https://imgur.com/a/iTmts
Album of three images so scroll down

Also here is a screencast of our application showcasing most of its features without having to run it from Xcode and deploying our server code:

Video #1: https://vimeo.com/158789199
Video #2: https://vimeo.com/158789218