Duke Shares Lunch

CS316 Group 22
Paul Dellinger, AJ Eckmann, & Josh Romine
11 December 2019

The Problem

Before each semester, every Duke student living on campus must purchase a food plan through Duke Dining, allotting them a set number of food points ($1 = 1 food point). While a student always has the option to purchase more food points, there is currently no way to return them – students who don't spend all their points must watch them expire at the end of the spring semester. Often, students with excess points will end up purchasing meals for their friends (or dining out at the Washington Duke Inn (no longer allowed)) but still have points left over. Entire vending machines have been bought out in the last week of the spring semester, because it's better than letting the money disappear -- Duke Dining keeps the profit and students have no way to recoup the money.

On the flip side, students who reside off-campus (such as many seniors) and don't have a food plan could stand to utilize others' excess food points. In addition, there are students who purchase food plans that are too small for their needs and run out of food points early find themselves purchasing all of their meals with real money, even as there are already-paid-for food points out there going unused. Included in this group of students are the freshman who only have $800 food points to get them through the semester. These groups of students will pay for their meals using cash or credit card at the end of the semester once they run out of food points. Both these groups of students could save money with an app that allows them to use other people's excess food points at a fraction of the cost.

Related concepts

There are numerous apps that use roughly the same concepts as our app. A phone app like Uber is looking to pair a driver (seller) with a rider (buyer). The driver can decide when and where he wants to drive and once a rider requests to ride, the driver then gets the opportunity to accept or decline the ride, and then the transaction is calculated and made from there. Our app is looking to do something very similar by pairing someone who is looking to sell food points with someone who is looking to buy food, while facilitating a transaction process. Uber

has the advantage of being able to handle the transaction themselves in-app. On the other hand, we simply connect them to a third-party, Venmo, to handle the transaction (though we pre-populate the amount field for the buyer).

Our system

PSQL Database

Our database has two schemas, public and basic_auth. The public schema has four tables:

RegisteredUser (uid, email, name,venmo, major, dorm)
ActiveSeller (saleid, uid, ordertime, status, percent, location)
Purchase (pid, saleid, bid, price, approve, paid, p_description)
SellPreferences (uid, location, percent)

Also defined is a view, ActiveRestaurants, which gives the locations of restaurants with active sellers and the count of how many sellers at each. These tables exist in the public schema, and are defined in build-db.sql. Also defined is a BEFORE trigger called no_rate_change that removes corresponding tuples in Purchase when tuples in ActiveSeller are removed or modified. Finally, a function called seller_purchases(sellerid) takes a uid and returns all the rows in Purchase whose seller matches that uid (requires a subquery to the ActiveSeller table).

The basic_auth schema has one table: Users (email, password, role). It has several functions which facilitate creating and logging in users. Most important is the login function, available to any anonymous request. It returns a JWT token that authorizes future requests for twenty four hours. The make_user function creates a tuple in basic_auth.users. The passwords in the users table are encrypted and decrypted using triggers.

There are three created roles: web_anon, authenticator, and todo_user. See the postgREST section for more info on roles.

PostgREST

PostgREST is an application that turns a PSQL database directly into a RESTful API. Tables in public are exposed to the server and can be accessed by making requests to links like so http://EXTERNAL_IP_ADDRESS/table_name

Different queries can be executed by adding parameters to the link.

when querying the Purchase table with GET /purchase we get this JSON array in return:


```
[
    {
        "pid": 2,
        "saleid": 1,
        "bid": 1,
        "price": 15.39,
        "approve": true,
        "paid": false,
        "p_description": "Meatballs & Spaghetti#Chicken
Alfredo#:notegoeshere"
    }
]
```


This is nice, but we'd like to have info on the buyer to display, without putting extra resources into another whole request on the RegisteredUser table. We also would like to get info on the seller, which would require a query on the ActiveSeller table with saleid and *then* RegisteredUser. We could write functions for the endpoints, which do all the queries, but PostgREST has this functionality built in:
GET /purchase?select=buyer:**registereduser(uid,name,venmo)**,
seller:**activeseller(registereduser(uid,name,venmo))**,pid,
Price,saleid,approve,paid,p_description

Which returns this JSON:


```
[
    {
        "pid": 2,
        "price": 15.39,
        "saleid": 1,
        "approve": true,
        "paid": false,
        "p_description": "Meatballs & Spaghetti#Chicken
Alfredo#:notegoeshere",
        "buyer": {
            "uid": 1,
            "name": "Paul Dellinger",
            "venmo": "paul_dellinger"
        },
        "seller": {
```

```
            "registereduser": {
                "uid": 1,
                "name": "Paul Dellinger",
                "venmo": "paul_dellinger"
            }
        }
    }
]
```

So in one endpoint, we've gotten info from the Purchase, ActiveSeller, and RegisteredUser tables. This approach of using the built in postgREST parameters does not work for every query however. For instance, to set up the ActiveRestaurants page we used a view, as there is no support for aggregate functions. Another example is getting all the rows in purchases associated with a certain seller. PostgREST doesn't support filtering based on a subquery, so we had to implement a function to do it:

```
CREATE OR REPLACE FUNCTION seller_purchases(sellerid integer)
        RETURNS TABLE (
        pid integer,
        saleid integer,
        price DECIMAL(5,2),
        approve BOOLEAN,
        paid BOOLEAN,
        p_description VARCHAR,
        buyername VARCHAR,
        buyervenmo VARCHAR
)
AS $$
        SELECT pid, purchase.saleid as saleid, price, approve,paid,
p_description,buyer.name AS buyername,buyer.venmo as buyervenmo
        FROM PURCHASE
        LEFT JOIN REGISTEREDUSER as buyer
        on PURCHASE.bid = buyer.uid
        LEFT JOIN ACTIVESELLER
        on PURCHASE.saleid = ACTIVESELLER.saleid
        LEFT JOIN REGISTEREDUSER as seller
        on ACTIVESELLER.uid = seller.uid
        WHERE seller.uid = sellerid;
$$
LANGUAGE 'sql';
```

PostgREST is still helpful in these cases, though. Any views in the public schema are automatically served at http://EXTERNAL_IP_ADDRESS/view_name
and any functions at http://EXTERNAL_IP_ADDRESS/rpc/function_name?parameter=input

Roles and Authentication

We have created three roles for regular use on the database.

- todo_user - the basic user, has permissions to tables in the public schema, must have authentication
- web_anon - any user not authenticated
- authenticator - a chameleon user that can become other users based on the authentication given

This is the set up recommended by postgREST. It allows you to set a password in the .conf file, and then any request can specify its role in the body of a JWT token. The token's authenticity is verified using the password. Once verified, the link is converted into a query and executed with that role's privileges. More information on the postgREST role system is available here.

iOS App

Login

- Users can login with a valid email and password. This creates an active token, valid for 24 hours before expiring (in which case one must log in again). Logging in also saves the user information so the user will not have to log in again.
- On the first visit to the app, one can tap Create Profile. This leads to a new view with fields to populate with name, email, password, Venmo username and optionally major and dorm. If the fields pass some conditions, tapping submit creates the user in the database, logs them in, and takes them to the home page.

Main App

There are three navigation controllers, accessed by tabs on the bottom of the screen.

- Buy: lists all entries of ActiveSellers table
    - Tableview with the locations at which people are selling and the amount of sellers at each location

        - Tap a location, and another tableview with each entry listing all the sellers at that location: their name, the food point rate (e.g. 5% off - a $10 purchase in food points would cost the buyer $9.50), and time to order.
    - A user can submit a purchase request, inserting a row into Purchases. Tapping on a sale for a restaurant shows the user a table for the menu for the restaurant. They can tap items and add order instructions into a text field. The order, total price, and buyerID, as well as the saleID are a part of the Purchases row. In addition, the entry has approval and paid columns, with default as False. After a request, the buyer is taken to a Waiting for Approval page (with a Cancel Order link to return them to the root buyer controller). Upon approval, they are taken to a new screen with a deep link to the Venmo app, pre-populating the price field (from Purchases table) and recipient (from Users table via Purchases-sellerID). A declined order triggers a pop-up notification of decline and returns the buyer to the root buyer controller.
- Sell: Has a tableview with three sections:
    - Your Sales
        - lists entries of ActiveSeller table that match your sellerID
            - each entry (one per location) includes a badge totaling the purchase requests for that sale
        - to make a sale entry, a user taps the Sell at More Locations. This takes them to a page to select restaurants to sell at (or simply toggle the All [West Union] WU Locations). They then select their sale rate and order time (they can alternatively select “I don’t care when I order” if they plan on staying in West Union for an extended time). If they select to sell at a location they are already selling at the row is updated in ActiveSeller
        - this page also includes a Cancel all Sales to remove the entries from ActiveSellers
    - Waiting for Approval
        - lists entries of Purchases with a corresponding row in ActiveSeller table that matches your sellerID and have not yet been approved. Seller can tap on sale to approve or decline the sale, or just swipe to delete (decline). Declining a purchase deletes it from the purchase table
    - Waiting for Venmo
        - lists entries of ActiveSellers table that match your sellerID, have been approved, but have not yet been venmoed for (Venmo column in Purchases is False). Once the seller has received Venmo, they tap on the sale and tap “I have received venmo.” At this point they order the food and give it to the buyer.

        - A final page shows the order instructions again, and a button to complete the purchase. Upon completion the purchase is deleted from the table

- Profile
    - Displays User fields name, email, password, major, and dorm.
    - Includes a button to log out, which removes the saved user information

Evaluation

We created a sample production database with 10,000 randomly generated tuples in registereduser, activeseller, and purchase. Despite the larger size, all response times were under 1 second on Postman, and most were between 300ms-500ms. Part of that time also accounts for the time it takes to transfer JSON objects of that size in an HTTP request. We simulated a user with 10,000 purchase requests, and saw virtually no lag on that user's profile on the app. With 100,000 purchase requests we saw a slight lag in displaying the table, but once loaded it ran perfectly. The sample dataset can be loaded by running

```
psql lunches -af sql/random-db.sql
```

Our overall implementation is sloppy and probably has more mistakes than we realize. This was our first time setting up a web server and using the Swift programming language, so there are definitely mistakes and stylistic issues. Despite that, the app generally works as planned.

Unresolved issues and future tasks

There are several ideas in our original idea that we did not have time to implement in this version of the app. Highest among these are sell preferences and filtering sellers by major and dorm. The schemas exist, though, so all that needs to be done is to design the viewcontrollers to facilitate the interactions.

With the Google Cloud credit, the app has cost us nothing to develop so far (other than time). However moving forward we may want to invest in some features that would make it feasible to move the app into production.
The process of getting the app on the app store is another hurdle we hope to get over sometime in the next view months. It costs $99 just to join as a developer, and then once

submitted it takes several weeks for approval. The app will have to be cleaned up and the UI improved before we can even get to that stage, however.

Another feature we plan to implement are push notifications. This would require joining the Apple Developer program. Push notifications depend would depend upon our server sending requests to Apple's Push Motification service (APNs). Our preliminary idea for this would be to configure triggers to send requests to the service. Our current implementation has no notifications and relies on pull to refresh and automated refreshing to give users new info from the server.
Right now the app is pretend secure. Our login process works fine and generates a valid JWT token for a user, however the `make_user` endpoint relies on the app having a valid, non-expiring token hardcoded in. Ideally we would use some sort of one time password for creating a user. Another possibility is integrating our authentication process with Firebase authentication. There is a lot of support in XCode for creating users, logging them in, and preserving user state once logged with Firebase, but it does cost money.

We currently have very little defense against malicious users. There are some regex expressions to filter out bad form inputs, but several areas are definitely lacking. We will definitely want to upgrade our nginx configuration to https. Right now, all a user would have to do is track their phone's network traffic to discover our database endpoints, look in the app data for the hard coded JWT token, and they'd have the power to delete the whole database. We would like to implement some amount of Row Level Security. In a preliminary look, row level security seems relatively simple to implement using Firebase authentication.