VennCafe

A Senior Project
presented to
the Faculty of the Computer Science Department
California Polytechnic State University

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science

by

Daniel Johnson, Christopher Clark, & Jonathan Amireh

Senior Project Final Report

Daniel Johnson, Christopher Clark, Jonathan Amireh
VennCafe, Fall 2015–Winter 2016
Faculty Advisor: John Clements

Project Overview

Our senior project involved creating a simple dating application and service. From other dating applications, we observed that the logistics of scheduling a first date are a tedious way to start a conversation with someone you've never met. The main concept of our application was that it would use user schedule availability and their favorite cafes to automatically plan optimal dates. Daniel titled it VennCafe because he likes silly titles, he likes venn diagrams and the .com domain name was available.

Initial Goals

We started the project with the intent of...

- Working through the entire development cycle of a large project as a team
- Devising techniques for determining overlapping preferences, especially when handling large amounts of location and time data
- Exploring different languages, tools and frameworks, especially involving web front-ends and native mobile apps
- Gaining a brief glimpse of what it would be like to be an entrepreneur promoting a new product

Deliverables

We performed the following tasks to complete our senior project.

- Created an initial requirements document and development schedule
- Deployed a publicly accessible back-end (developed by Chris and Daniel)
- Created a web front-end (designed by Daniel)
- Created a native Android app (developed by Jon)
- Assembled our team's thoughts and user feedback into this final write-up

Technologies Used

This section contains the list of the tools, languages, libraries, frameworks, services and other technologies that we incorporated into our project. Because our end goal was to create a finished user-facing application, a large portion of our project involved determining which existing technologies would save us the most development time. For each tool, we include our initial reasoning for using it, as well as our thoughts after using it throughout the project.

Back-end

Node.js and Express

We chose to use Express, a lightweight back-end framework, because VennCafe had only a few endpoints but a lot of custom functionality. We thought that the custom functionality would be difficult to implement in a more opinionated framework. After using it, we found that Express did allow us a lot of flexibility, and we didn't have to work around its design decisions to implement any of our features. However, because Express did not enforce any specific design patterns, we had to put a lot more effort into structuring our code and installing middleware. Sometimes, if we implemented sections of the application without thinking about the higher-level design, we ended up having to refactor it later avoid code repetition.

Using JavaScript on the back-end was usually fine, though we encountered more runtime errors than seemed necessary. Using a compiler (e.g., Google Closure Compiler) or type system (e.g., TypeScript) would have been worthwhile.

Sweet.js and cspjs

Using Express requires using asynchronous functions that accept callback functions, which leads to callback hell when used without a control flow library. Some solutions include the npm module "async," promises and generators.

For our control flow, we used the npm module "cspjs," which included macros written in Sweet.js for calling asynchronous functions. The benefit was that the concise syntax made our code look much cleaner. Overall, however, it didn't seem production-ready. Sweet.js had some issues with outputting correct sourcemaps when using modules, which made it difficult to debug the original source code or calculate code coverage. cspjs would silently fail if used within an unsupported control structure (e.g., a "for" loop). Additionally, supported control structures could be inconsistent; in one case, the "catch" syntax failed due to the presence of an "if" statement, but a similar alternate syntax was used successfully. If we used modules that didn't use Express-style (err, value, …) callback functions and didn't think about how cspjs worked internally, it was easy to accidentally write code without realizing that thrown errors would be lost. For example, cspjs implicitly adds a callback parameter to its function headers, then catches errors and passes them to this callback function. Our testing framework's "test" function passed its callback function as a field within an object, and cspjs didn't know how to access that callback method to pass on the error.

After trying a few options, we think that promises are the cleanest and most reliable control flow method currently available. In the future, especially with newer ES7 syntax and transpilers, other options may improve and are worth revisiting.

PostgreSQL

We chose to use SQL, as opposed to a non-relational alternative, because all our data had a standard schema. We chose to use Postgres because it is a well-established DBMS, and installing PostGIS made it simple to add an index based on a user's latitude and longitude. For the most part, using Postgres worked well.

Using SQL from Node.js felt a bit unnatural, because we couldn't find a useful ORM module or database abstraction layer with npm for our specific use case. For our application, we didn't need to perform very many distinct actions with the database. The actions we did need to perform, though, were usually complex SQL queries with several joins, and it was easiest for use to manually write these SQL queries. We tried a few ORM and query builder modules, which were helpful for simple use cases, but were inconvenient to use with the more complex queries. We wrote our own simple database abstraction layer, which was a convenient way to associate manually-written queries with our predefined data model types. These SQL queries were parameterized to prevent SQL injection, using placeholders for the data that would be available in a given instance of the specific data model type. This approach worked well, but if our application's features got larger in scope, we would likely have to use a module to automatically write common simple queries. It would have been worthwhile to look into some ways to integrate an existing npm module with our custom queries.

It's unclear how well Postgres would scale to handle more users and a more complex application. Introducing a sharding strategy with Postgres would be somewhat difficult to set up, especially since our application is designed to pair multiple users together, potentially across multiple shards. It may have been worth looking into something designed around scalability, such as Cassandra or MongoDB. We also ended up with some very large queries for determining which users to pair together. We're not sure how we would do something more difficult within that query, such as joining a result set with lazily-loaded data from the Google Places API. If we continued adding more complex features and user preferences, we would eventually need to find a solution to this problem.

External APIs

Facebook

The login API was fairly straightforward, but the documentation could have been better. The process of building a backend login workflow with their API was well-explained, but the individual endpoints were somewhat glossed over. Some endpoints referenced contained links to their expanded documentation, but most did not. The JSON response formatting was also somewhat inconsistent. In particular, the endpoint for inspecting user access tokens returned a JSON object where all fields were inside a nested object labeled "data," while all other endpoints we used returned fields in the top-level JSON object in the response. This was noted in their documentation, but seemed somewhat unexpected. The only other endpoint with this structure referenced in their login API documentation used the "data" field to hold an array of objects, but

it was not indicated that the endpoint we were using might return an array instead of a single object. All things considered, the API behaved as expected; these are merely minor comments.

We had to include a privacy policy for Facebook to approve our application, which we generated using a script from a GitHub user. It's still unclear to us what we actually need to include in a privacy policy, which makes us wonder how much legal advice is needed to launch a new application or service.

Basic Facebook profile information was available to our application by default, without requiring our app to go through a review process. Specifically, we used their API to retrieve a user's first name on account creation. It would have also been desirable to retrieve their birth date, but this would have required our app to be reviewed by Facebook, which did not seem to be feasible to attempt before the end of our work on this project.

Google

Google Places API Web Service

Overall, the Google Places Web Service (HTTP API accessible from our back-end) was well-documented and easy to use for simple use cases. The only difficult use case was retrieving the open hours for a list of places. We had to perform one request per place to get its open hours, which was slow enough for large lists that it was unusable. As a next step, we would have implemented some form of caching.

Google Places JavaScript API

The front-end library was straightforward to use. However, we did have to write a wrapper around its callback-style functions to use it easily from within AngularJS. Some cafes had incomplete data, such as no photos, missing open hours, or were poorly categorized. We could have tried out the Yelp API to see if it had more reliable data for photos and categories, but the Yelp API did not include open hours at all.

Front-end

Sass

Sass was easy to set up, especially since the SCSS syntax worked with our existing CSS. It had a lot of useful features, which helped us structure our SCSS without much boilerplate or repetition. I don't think I'll ever write plain CSS again.

AngularJS

AngularJS worked very well for creating web UIs quickly without much development time. It's a very opinionated framework, and a lot of magic went on behind the scenes. Because I had to rely on Angular's features for many tasks, I think that it would be difficult to use Angular if support for the framework ended. I would be concerned about using it in a production product until after the dust settles during the

Angular 2.0 release, and it might be worth using a simpler framework in production products. Overall, I was impressed with how well the framework worked, and will definitely revisit it after the 2.0 release.

Android

Kotlin

For part of the development process, we used a Java interoperable language called Kotlin to see if an improvement could be made beyond Java's normal syntax. Transition to the language was easy due to the fact that the Kotlin team included a module that automatically converted Java source code to Kotlin.

Syntactically, Kotlin is shorter than Java 8 with native support for lambdas, custom getters/setters on fields, null-type safety, default parameter values, extension functions, and much more. In general, it was more pleasant and much quicker to write features in Kotlin rather than in Java. The Android community seemed excited to introduce the new language into the development cycle. Square Inc's head developer, Jake Wharton, released an internal document from their development team explaining why Kotlin, even its beta form, was ready for use in production applications.

Kotlin seemed like an incredibly promising fourth-generation language offering many of the amazing syntactical features of Swift in a lightweight, JVM compatible language. Theoretically, it should have been completely compatible with all Android features and notions however we ran into several issues along the way.

Firstly , Kotlin support in terms of Android Studio came in the form of a plugin which was updated at the same time as language releases. Since the language was not natively support by Intellij (yet), the autocomplete features that a developer could normally rely on for quick documentation lookup and general syntactical help, was incredibly slow. On average, we found that it took the plugin about 7-10 seconds to load autocomplete options versus Java of about 1-2 seconds. On top of this, the time to deploy was close to 45 seconds per build under Kotlin while it about 20-30 seconds on Java.

The Gradle build system is the primary way of building any Android project since the switch to Intellij-based Android Studio. Recently, the Android Tools team along with the Gradle team have noticed that the Gradle build system for Android has grown sluggish and slow and have attempted to integrate several features to try and speed up the process. One of which included incremental building where only source files that could be mapped to their class files were included in the final APK. Under normal circumstances (that is, a Java codebase), this would have been a perfectly acceptable change. However, during development, we ran into NoClassDefFound errors that seems to be random until further testing revealed it only occurred when leaving intermediate build artifacts between builds. Discussion on the Kotlin Slack chanel revealed that this was related to an open bug, KT-10733, where Gradle is specifically looking for *Java* source files when performing incremental builds. This caused class files generated by Kotlin to be tossed out during builds resulting in said NoClassDefFound errors.

Finally, there were minor annoyances such as the Intellij debugger not adapting properly to the resultant bytecode when stepping through Kotlin code. For example, if a return statement was reached in the middle of a function, the debugger would step beyond that line visually while in the actuality, it's stopped execution for that function as expected. Considering all of these factors, especially the Gradle build compatibility, we chose to stick to a Java codebase while using libraries such as Retrolambda to backport Lambdas from Java 8 into Android's Java 6 environment.

ReactiveX

ReactiveX is a set of libraries based on the concept of reactive functional programming. This primarily refers to event-driven data streams with functional programming aspects like map() or filter(). Specifically, this project utilized the Java module for this system, RxJava, for callbacks on network requests made to the API. RxJava allows for easy background threading by switching between different Schedulers with one-line of code.

In addition to the Java module, we imported an Android-specific package called RxAndroid in order to schedule tasks on Android's UI thread. This was necessary because the RxJava module is a JVM-only module and did not include a Scheduler specifically built for Android. This mainly allowed for network requests on a background thread and responding to the response by updating on UI thread with relevant information.

RxLifecycle was one last Android package that allowed for any data stream's life in memory to be tied exclusively to an Android Activity, Fragment, or View. For this project, this would have allowed for network requests to "die" when the user navigated away from the current screen and prevented the requests from continuing if there's no UI to respond to the result of the data stream. However, this module went unused due to time constraints.

Deploying

AWS

We deployed our code to an AWS EC2 instance we set up, mostly because we wanted to learn about system configurations. Our first impression was that the AWS web interface was difficult to use, and something like DigitalOcean may have been friendlier. Setting up the web server was not too difficult, but it would require some effort to maintain and monitor (installing security updates, checking logs, configuring firewall rules, making sure we don't run out of storage space, scalability concerns). A PaaS solution such as Heroku would be a possibility if we didn't have time to maintain the AWS setup.

Let's Encrypt

We needed to send user session data over HTTPS (HTTP over TLS) to prevent session hijacking, among other privacy concerns. We needed a TLS certificate signed by a certificate authority to allow clients to authenticate the identity of our server. Most of the time, getting a certificate signed by a certificate

authority is not free. Let's Encrypt is a new certificate authority currently in beta, and using it is free and automated.

Let's Encrypt was surprisingly easy to use, aside from some issues with requiring root privileges. We just ran one command and then edited our NGINX configuration. At the time of writing, the only hassle was setting up a script to automatically renew expired certificates every three months.

NGINX, pm2

We set up NGINX to work as a reverse proxy to pm2, which is a node process manager. We used NGINX to handle basic tasks, such as gzipping static files and using TLS, and all our business logic was in our node application. This worked well except for some extra configuration to send secure cookies to our application ('trust proxy' in Express).

Workflow

We used a remote git repository (Bitbucket) to share our code. We used shared Google Drive documents for to-do lists and stored some API documentation in Markdown files in our repository. We talked about ideas in person and over chat. It worked reasonably well, except for a few times when we changed something without realizing one of us was relying on it working a certain way. Our workflow would likely not scale to larger teams.

One of us caught mono during development (not as a result of using the application), which led to decreased time for development, and less availability for talking about sections of code that only that person was familiar with. The main takeaway from that is that life events can slow down application development, and it would be worthwhile to have at least two developers are familiar with each section of code.

Main Issues

Pairing Users

Main Query

The main function of our application was pairing two users together at a specific time and place. We selected this data with a large SQL query that joined most of the tables in our database, assuming that the DBMS would run faster than if we used our own iterative implementation. We tested the performance of our query, and joining user schedules together was the most time-consuming operation. Overall, however, the query would run quickly because the PostGIS spatial index would filter out users that were not in a nearby location.

Optimistic Locking

One design decision of our application was that a user could only be matched with one other user at any given time. This created the possibility of a race condition where two users could end up matched with the same third user if the matching operation was asynchronous. Doing the entire operation at once was either too slow or caused deadlock if we weren't careful about which rows got locked in different orders by simultaneous UPDATE queries. We ended up using an optimistic locking strategy, which would rollback and retry the entire operation if the relevant data happened to get modified in the middle of the operation. The idea was that we increased the speed of the operation under the assumption that collisions were unlikely (two users with nearly identical preferences in the same location clicking the same button at the exact same second). The solution felt like ugly code, essentially "retry until it works." However, it performed better than our other solutions, even with nearly simultaneous usage from multiple users.

Transactions

A common issue involved the use of transactions. Sometimes we would need to wrap several queries in a transaction, in case we needed to roll them back at some point. Some functions would start a transaction, run a few queries, then commit the transaction. If a function like this called another function like this, the application would request a new transaction before releasing the previous transaction, which could lead to deadlock with the maximum number of connections. To fix this problem, we would need pass information about database connections to every function called from an individual user's request, rather than allowing any isolated function to start a transaction. The transaction functions in node-pg-db appeared the solve this problem, but we avoided them because they used domain, a deprecated node module.

Timezones

One challenge we encountered was translating from a user's preference, such as "I'm free at 4pm on Wednesdays," to an exact time with a specific timezone. We opted to include timezones as soon as possible, so all schedule preferences were saved with a UTC offset (timestamptz column in our database). This decision made it simple to calculate a time with a timezone for planned dates. The main issue involved calculating recurring schedule preferences over daylight saving time. For example, if we took a time with a UTC offset and incremented it by one week, it would be unaware of any changes to the UTC offset for corresponding timezone, and could be off by one hour. We had to autodetect the full timezone name on the front-end and save that string to the database, then use Postgres's "at time zone" syntax to increment timestamptzs by the correct amount.

Testing

It was difficult for us to find time for both development and testing. Integration tests on our API endpoints ended up being the most effective, in terms of bugs caught versus development time required. Unit tests were useful, but tedious because we had to mock a lot of dependencies. Structuring and standardizing our tests better would have helped us spend less time writing more tests.

The most effective integration test was a single test that created 100 randomized user accounts simultaneously and had each of them request a matched user simultaneously, then verified the preferences of matched users. This test would catch most race conditions and incorrectly-used database transactions, but it took some debugging to figure out where each bug was actually located. We also included some performance tests, which would alert us if we altered our SQL queries in a way that prevented them from using the correct indexes.

We didn't test much of the UI. The UI was changing a lot during development and using a tool like Selenium WebDriver was more work than manual testing. The front-end didn't have much business logic, aside from some date and time manipulation on the calendar preferences, which prevented unit tests from being useful.

Security and Privacy

We used Facebook Login and sent session cookies over HTTPS. We validated user input and used parameterized queries. We did not save any user information that the user didn't manually enter, and other users would only see other users' information as necessary. For example, we rendered birthdates to ages on the back-end, and Facebook user IDs were mapped to our own UUIDs. In an attempt to not expose too much location data, we estimated user locations based on the manually inputted places that the user indicated he or she would be willing to travel to.

Locations

Each user's location was determined by aggregating the geocoordinates of the Google Places they selected as preferential date locations. This was partially a security measure, as mentioned previously, and partially a component of user experience, as it seemed correct to assume that users would want to match based on where they want to meet, not where they are currently located. The date locations were stored as Google Place IDs, which could be sent to Google's API to get expanded information such as geocoordinates and hours of operation. Because making external API requests over HTTP is slow, and user matching is a frequent operation, we elected to store a pre-calculated user location and match radius for use in the matching query. These pieces of location data would only be updated when a user updated their preferred date locations (an operation which we expected to be much less frequent than matching). The user's location was calculated as the average point (in latitude and longitude) between the geocoordinates of all places they selected, and a match radius that encompassed all their places accompanied their central location. We used PostGIS to create an index on the user's location, which was crucial to the performance of the match query. Because of the importance of the spatial index to match query performance, a maximum match radius was hard-coded into the location API handler. This maximum value was selected somewhat arbitrarily based on Chris' perception of a reasonable distance to travel for a date, and could surely be improved upon by research of user experience and match query performance.

Genders

For the sake of social consciousness, we decided to not restrict users to a gender binary, and to allow our application to be extensible with regard to gender preferences. This required the addition of a few more joins to our match query, but was not a performance bottleneck in our testing (after proper indices were created). Our set of genders is still static with regards to the global application, and can't be customized by either API or end users, but could be more easily updated in the future if users so wished.

Request Validation and Error Handling

Request Validation

We largely relied on validate.js for request validation in the API. This library provided a declarative pattern-matching approach to request body validation, which was very clean, readable, and easy to update as needed. This library provided a good set of validators for simple use cases such as checking numbers and dates, but its primary advantage was the ability to extend it with custom validators. It also had the benefit of providing customizable validation failure messages, which could be passed to our error system.

Error Handling

We created a simple error object that included some logging functionality (printing to console) in dev environments. We also created an API endpoint factory that would intercept any errors thrown in the API endpoint handlers, and send an appropriate API response. This mainly consisted of sending HTTP status code 400 (to indicate a bad request) for certain error types, and status code 500 (to indicate server error) in all other cases. In development environments, any error message would also be sent as part of the API response. Our error system was useful in our development, but was not ready for a production environment. Ideally, we would have liked a persistent logging system that could be checked for details of any error that occurred. Additionally, our API endpoint factory only handled errors that were the result of our own endpoint handler code, but errors encountered by middleware would not be handled gracefully by this system. This was an issue in a small number of cases, and led to more time spent debugging.

Mobile-friendly Website

Compared to developing a native Android app, it was much easier to implement a mobile-friendly version of the website with a few CSS media queries and some support for touch events. It didn't feel quite as smooth as a native mobile application, however. This makes the decision between native and web into a trade-off between diminishing returns in quality versus more development time. Rendering web views for some sections within a native mobile app could be a feasible compromise.

User Feedback

Toward the end of development, we collected user feedback. We created a survey using Google Forms, then displayed a link to the survey in a notification after a user filled in a profile and attempted to plan one date. The overall feedback was positive, indicating that users liked the concept of the app, thought it worked well, and thought that the user interface was intuitive. The critical feedback mainly involved requesting features that we had planned on implementing but had not finished. Many users wanted a better notification system. The application refreshed the user's notifications every time the user performed an action, but it would have been ideal to use a WebSockets library to automatically refresh the notifications as the events occurred. We would have also liked to use Twilio to send text message notifications. Additionally, some users wanted more control over profile preferences and their profile photos.

Some users said the application seemed to encourage going to locations that could potentially be dangerous, such as non-busy parks at night. To fix this, we'd need to tweak our usage of the Google Places API to select busier places, and we could possibly include warnings that users should be cautious when meeting people for the first time. We also considered the idea of allowing users to add a friend's phone number as an "emergency contact" that would receive an automated text message with the user's location at the time the date starts.

Some users indicated that they might want to plan multiple dates at once. The original reasoning behind allowing only one date at a time was to encourage users to just go ahead and meet each other, rather than spending time comparing people on the application. This original reasoning could be wrong, however, so we're interested in what users would think if they used it for a while with a larger user base.

Reflection

Main Takeaways

The main thing we realized was that this project took more time and effort than we originally thought. We didn't finish all of our features. We spent a lot of time on the calendar feature during the first quarter of development, then rushed some other features during the second quarter of development. This knowledge will help us choose appropriate scopes and development plans for our projects in the future. For example, this application included a few new ideas and interesting computer science problems to solve, but most of our time was spent solving problems that had already been solved, such as creating good user interfaces.

After deploying our application, we weren't very interested in promoting it and getting people to use it. We would show it to people, but mostly to demonstrate the technical features we worked on rather than the application itself. Daniel realized that he likes trying out new ideas, but is definitely more of a

developer than an entrepreneur. We didn't even think of any ways to make money off of this application, other than hoping that Tinder would like the idea and buy us out.

Future Work

In its current state, a link to this project is a good addition to our portfolios. If we wanted to promote the application and get people to use it, we would first want to do a little bit more development and cleanup, as indicated in some of the previous sections of this write-up. After that, we could start having more users test the application and make sure it scales well.