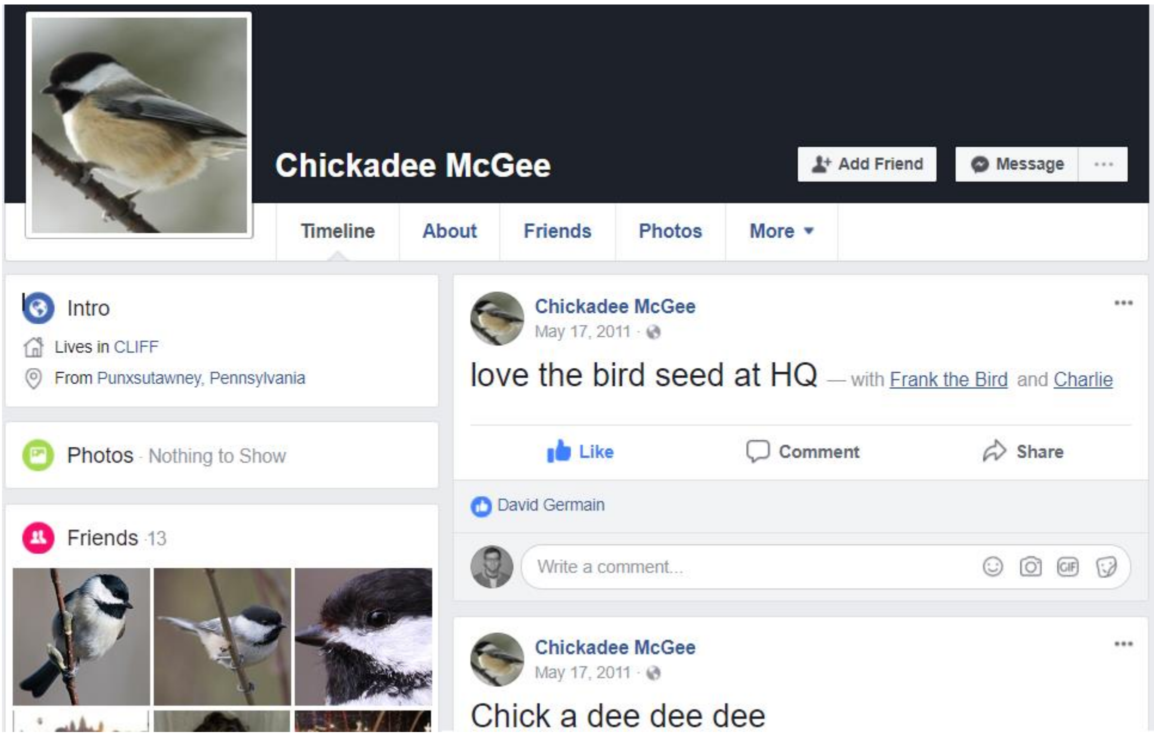


Facebook for Birds: Adding Statistics and Cool Numbers

NORTHERN MICHIGAN UNIVERSITY

David Germain | CS480 | 4/10/2018

Check out the website at jpcp.nmu.edu

Background

There are these birds. People want to study them and understand their behavior, culture, and buying habits. Just kidding about those last two. Some biology people have banded the birds with RFID tags around their legs. They also made special feeders that contain their favorite seeds and detect when a banded bird visits and records that to an SD card. Later, Jerry Connor and Michael Whalen upload the data to a database on Euclid. Then the basic API and website serves that up to users.

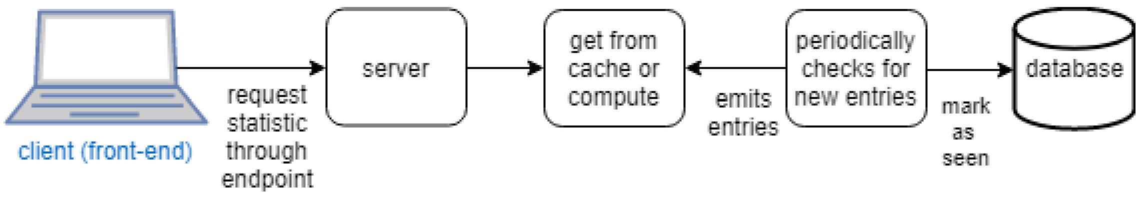
Introduction

The main purpose of the JP Chickadee Project website is to engage the community and make them curious and about birds. In addition, to get them excited and asking questions. To help achieve this for my senior project, I added cool graphs and numbers to the website and designed the accompanying backend.

I chose this project because I like statistics and handling large amounts of data is exciting. My near-term career goal is to work on web apps, so it feels like a good idea to have a sample of my work online to show potential employers.

Overview

There are several parts to this project. The main database, which I added a column. A thing that regularly checks the databases for newly added entries. A thing that does the statistics. A thing that caches computationally heavy statistics. A server that answers http requests. And front-end components.


client (front-end)
request statistic through endpoint
server
get from cache or compute
emits entries
periodically checks for new entries
mark as seen
database

Technologies

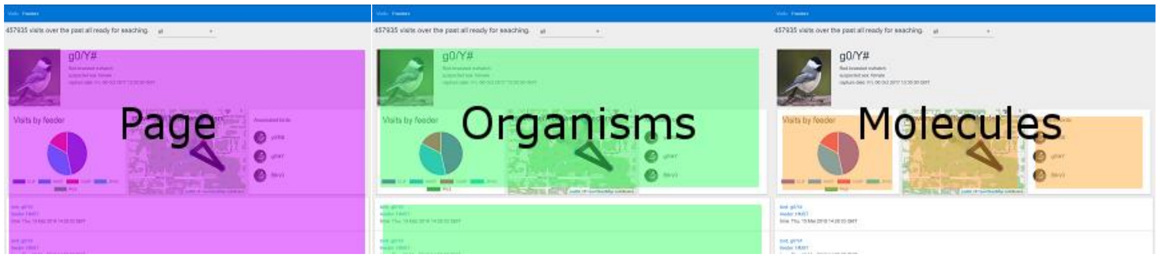
Since this project is part of a bigger whole, I want my code to be maintainable by other people. To me that meant trying to keep the technologies to learn to a minimum. In addition, when picking a framework or library to pick one with decent documentation.

I ended up using the following technologies:

Technology	Description
JavaScript	
Chart.js	
Leaflet	Library for displaying maps
Lodash	Library of nice utility functions
~~Pug~~	~~Make templates with variables that turns into HTML~~
Vue	Front-end component framework
Vuetify	Material design that works with Vue
HTML	
CSS	
Npm	Node package manager
Node	JavaScript runtime environment with lots of libraries
Git	

Front-end Structure

Components make up the front-end. Components are made of a combination of sub-components and HTML. The levels of components that I have used are pages, organisms, molecules, and atoms. Which I learned from here. For example, there is a bird profile page (below in purple). This is made up of a bird profile and a visit list organism (below in green). The bird profile organism is made up a visits pie chart, a movement map, and an associations list molecule (below in orange).



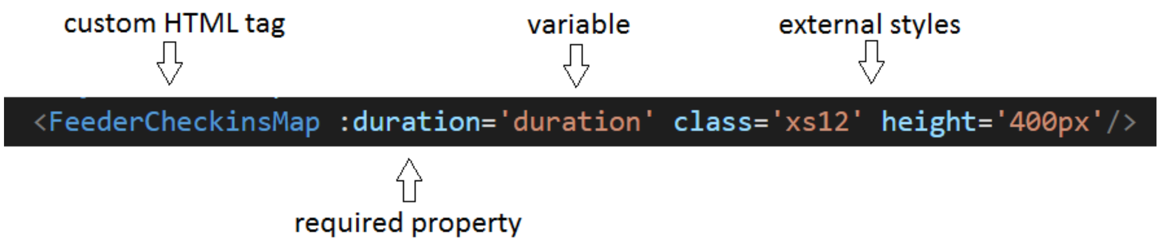
I had a hard time determining whether pages, organisms, or modules should be making the network requests. I still feel it is somewhat arbitrary, I picked molecules, and just consistently stuck with it.

A downside with having molecules make the network requests is the possibility of duplicated requests. If I have a pie chart and bar graph molecule that both hit the same API endpoint that is two network requests for the same data.

On my proposal, my grading rubric has points for making "modular components." What does that mean? To me that means individual graphs/components are self-contained. If you declare a component with the correct properties, it should just work and look okay.

A modular component needs to know:

- how to get the data it needs
- how to update when the properties change
- basic styling like min-width and min-height



```
<FeederCheckinsMap :duration='duration' class='xs12' height='400px'/>
```

Reducing Big Duplicated Network Requests

On the front-end two of my components, the leaderboard and associations list, need to display the birds' band combination (for example G0/Y# or #R/W0) and name (for example Karen or Tim). The problem is that my analytics API does not know about band combinations or names, to my API birds are simply their database id. That means the front-end has to get them from somewhere else. Band combinations are from the CRUD API by making a `GET` request to `/api/birds/`. Names are in a dictionary stored on the front-end.

The leaderboard and associations list components need to show band combination and name. Making that big request to `/api/birds/` each time a user views a bird profile or homepage was noticeably slow. I solved this issue by wrapping the endpoint in a class and making it a singleton. Now the birds are only requested once.

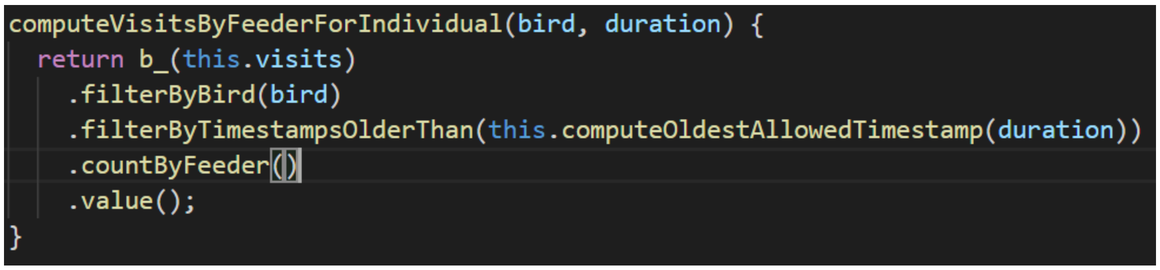
Since the website is a single page application, the singleton lives while the user switches from page to page and re-initializes on page refresh.

Back-end Structure

The backend consists of three main pieces. The server, statistics, and database wrapper. The server is a program that is always up and running. The database wrapper checks for newly added birds, feeders, and visits. We know if an entry is new because I added an isSynced column to each table that by default is false. The database wrapper emits the visits to the statistics object and they are appended to an array. Most statistics are calculated on request using the array. I made a cache object because calculating bird associations on every request was really slow.

Most Fun Part

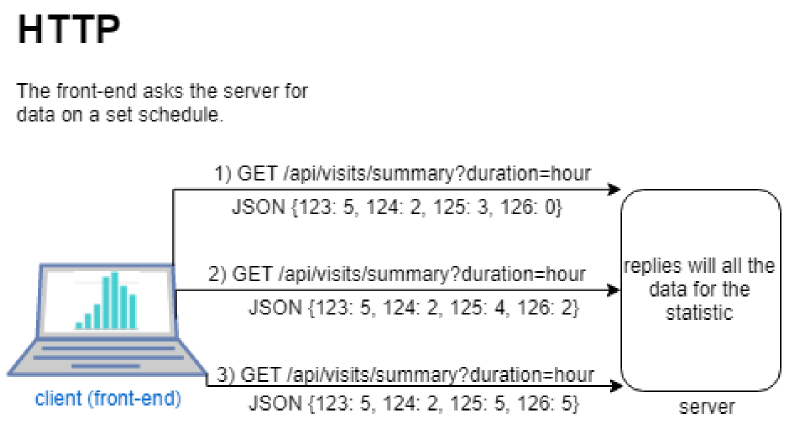
Lodash is a utility library for JavaScript. It is great for functional programming. It allows for method chaining and registering custom methods. To explain see the picture below, b_ is Lodash with my custom functions added like filterByBird, filterByTimestampsOlderThan, and countByFeeder.

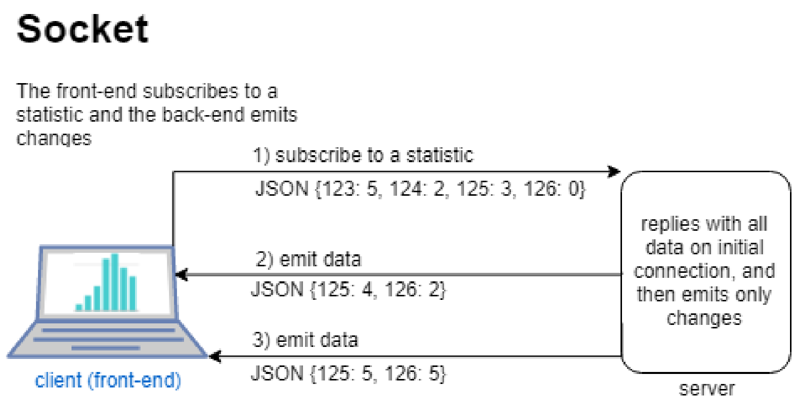


```
computeVisitsByFeederForIndividual(bird, duration) {
  return b_(this.visits)
    .filterByBird(bird)
    .filterByTimestampsOlderThan(this.computeOldestAllowedTimestamp(duration))
    .countByFeeder()
    .value();
}
```

Mistake #1

If I were to do this project over, I would have the front-end communicate with the back-end using sockets instead of HTTP requests. HTTP requests work okay, but for the “Updates without page refresh” option of my assignment it is not so good. In addition, I think using sockets would encapsulate the statistics better by keeping the rate of change close to where the statistics are. For example, a bird’s lifetime statistics probably does not significantly change by the minute, so with the current implementation the front-end needs to know how often it should refresh its data for each type of statistics.


HTTP
The front-end asks the server for data on a set schedule.
1) GET /api/visits/summary?duration=hour
JSON {123: 5, 124: 2, 125: 3, 126: 0}
2) GET /api/visits/summary?duration=hour
JSON {123: 5, 124: 2, 125: 4, 126: 2}
3) GET /api/visits/summary?duration=hour
JSON {123: 5, 124: 2, 125: 5, 126: 5}
replies will all the data for the statistic
client (front-end)
server

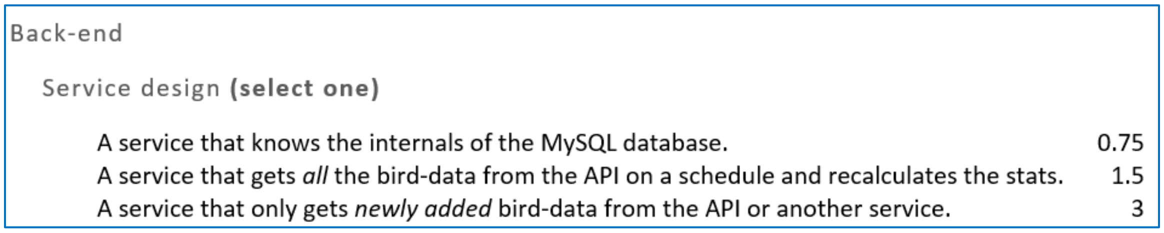

Socket
The front-end subscribes to a statistic and the back-end emits changes
1) subscribe to a statistic
JSON {123: 5, 124: 2, 125: 3, 126: 0}
2) emit data
JSON {125: 4, 126: 2}
3) emit data
JSON {125: 5, 126: 5}
replies with all data on initial connection, and then emits only changes
client (front-end)
server

Mistake #2

Pug is a like HTML but has templates and variables and during the build step, it becomes HTML. Originally, I was going to use Pug instead of html because it seemed nicer. However, I soon regretted it and switched because all the code samples for Vue and Vuetify use HTML.

Mistake #3

For the “service design (select one)” portion of my assignment, I feel a little confused about which point tier I achieved. See the tiers in the picture below.



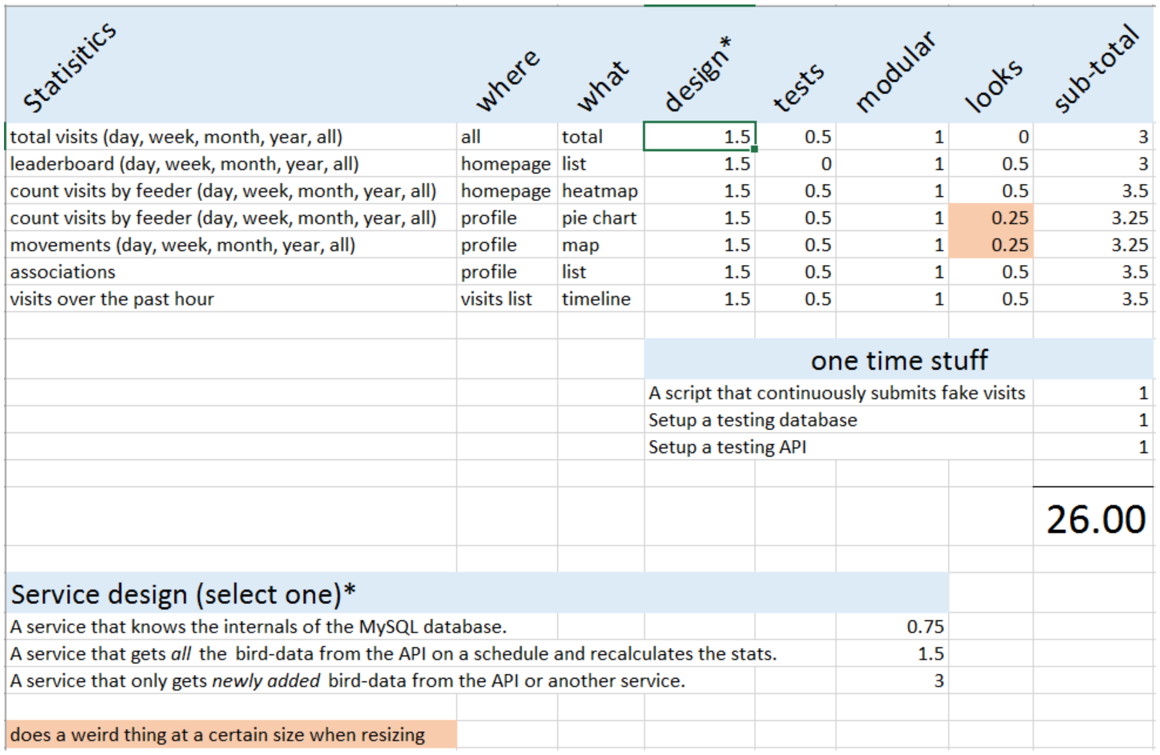
Back-end

Service design **(select one)**

A service that knows the internals of the MySQL database.	0.75
A service that gets *all* the bird-data from the API on a schedule and recalculates the stats.	1.5
A service that only gets *newly added* bird-data from the API or another service.	3

The database is behind a wrapper class that changes the representation of feeders and birds to just their database id and emits them as JSON. Therefore, I feel I achieved past the first tier. I think I am at the second tier because although I only get the newly added data, I recalculate the stats instead of doing some kind of cool updating. For example, a statistic of the ongoing visits, if I get one new visit I throw the total away and recount them instead of just adding one to the total.

Grading Rubric



Statisitics	where	what	design*	tests	modular	looks	sub-total
total visits (day, week, month, year, all)	all	total	1.5	0.5	1	0	3
leaderboard (day, week, month, year, all)	homepage	list	1.5	0	1	0.5	3
count visits by feeder (day, week, month, year, all)	homepage	heatmap	1.5	0.5	1	0.5	3.5
count visits by feeder (day, week, month, year, all)	profile	pie chart	1.5	0.5	1	0.25	3.25
movements (day, week, month, year, all)	profile	map	1.5	0.5	1	0.25	3.25
associations	profile	list	1.5	0.5	1	0.5	3.5
visits over the past hour	visits list	timeline	1.5	0.5	1	0.5	3.5

one time stuff	
A script that continuously submits fake visits	1
Setup a testing database	1
Setup a testing API	1

26.00

Service design (select one)*	
A service that knows the internals of the MySQL database.	0.75
A service that gets *all* the bird-data from the API on a schedule and recalculates the stats.	1.5
A service that only gets *newly added* bird-data from the API or another service.	3

does a weird thing at a certain size when resizing

Grading Scale

22≤	A
20	A-
18	B+
16	B
14	B-
12	C+
≤10	C

Lines of Codes

Frontend: 531

Backend: 517 (and 250 of unit tests)

Fake feeder: 43