CoB Transparency Project Final Report

97A Field Study
05/04/2020

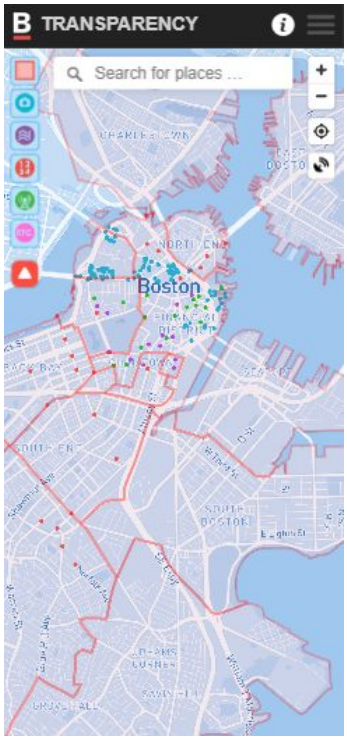
Belle Scott, bellemscott@gmail.com
Youbin Ahn, ybin.ahn@gmail.com
Rhosung Park, rhosungpark@gmail.com
Anyan Xie, anyanxie@gmail.com

1. Introduction

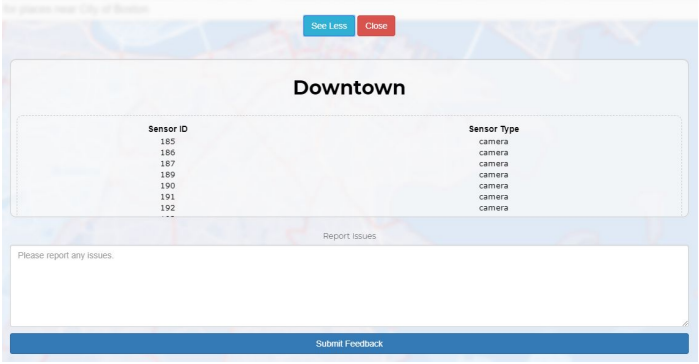
With an increase in sensors within Boston city, the City wants to explore the public's attitude towards various Internet-of-Things(IoT) sensing that is or may be deployed. The objective of the project is to generate more general awareness and transparency about the use of IoT and the data use, privacy, and security concerns that technology in the public realm may raise among users. Here we built a responsive web app, ***Transparency***, to educate the citizens of Boston on active sensors that exist within the city limits and to explore their attitudes. We also enable authorized site administration and provide easy-to-use interface for the City of Boston staff.

2. App Functionality

2.1 Sensor Map



To achieve the main goal of showing the sensor information as transparently possible, the map functionality of the application is purposely built to have a very simple view. Users of the application can click on each "sensor" (represented as dots of different colors) or "neighborhood area" to either read the information related to each element or leave feedback about them. When the users click on the sensor of their interest, an information container will pop up from the bottom of the application to reveal its data including the ***Sensor label***, ***sensor type***, ***sensor location***, ***sensor owner***, and ***sensor explanation***. When the neighborhood area is clicked, the same information container will slide up, but this time with different information comprising the name of the clicked boston neighborhood, and the list of the sensors within the neighborhood (Each row of the sensor list contains the sensor label, and the type of that sensor). The users can achieve the same behavior by using the "search" function of the application as well. For example, when the "sensor 11" is searched, the map will automatically move its view to contain that sensor in its center, and slide up the information container to show users the information about the searched sensor. Some minor functionalities of the map include toggling of each type of sensor, zooming in and out with responsive sensor size, converting the map into satellite view, and locating the current view to the user's location (geolocation).



The above features and explanation on how to use the app is also included in the app itself, and can be seen by clicking on the information button on the top right side. The expanded page for explaining how to use the app is shown in the below figures.

2.2 General Functionality/Use Cases

Since the app is to promote greater transparency toward the increasing IoT sensors within the city, the team thought that it would be inappropriate for the users to have to share their information or to share their locations to use the app. For this purpose, the app does not require any sign up for general use cases nor does it require the users to share their locations. However, sharing their gps location is added as an optional feature. When users decide to share their GPS location, the app will place the individual's exact location on the map and will help aid in seeing which sensors are around that particular individual. However, in order to provide a similar feature, without having to share the locations, we have included the search bar at the very top. This similar task can be achieved by entering the street address the individual is at, which will then fly to that location on the map. Not only that, the searching feature allows users to search for sensors around a specific location before they visit the location.

2.3 Admin Functionality

In order to help future administrators of the app navigate the rails admin tool, we built a comprehensive manual that contains all the necessary information administrators need to use the tool, including tutorials to import, export, add, delete, and edit sensors.

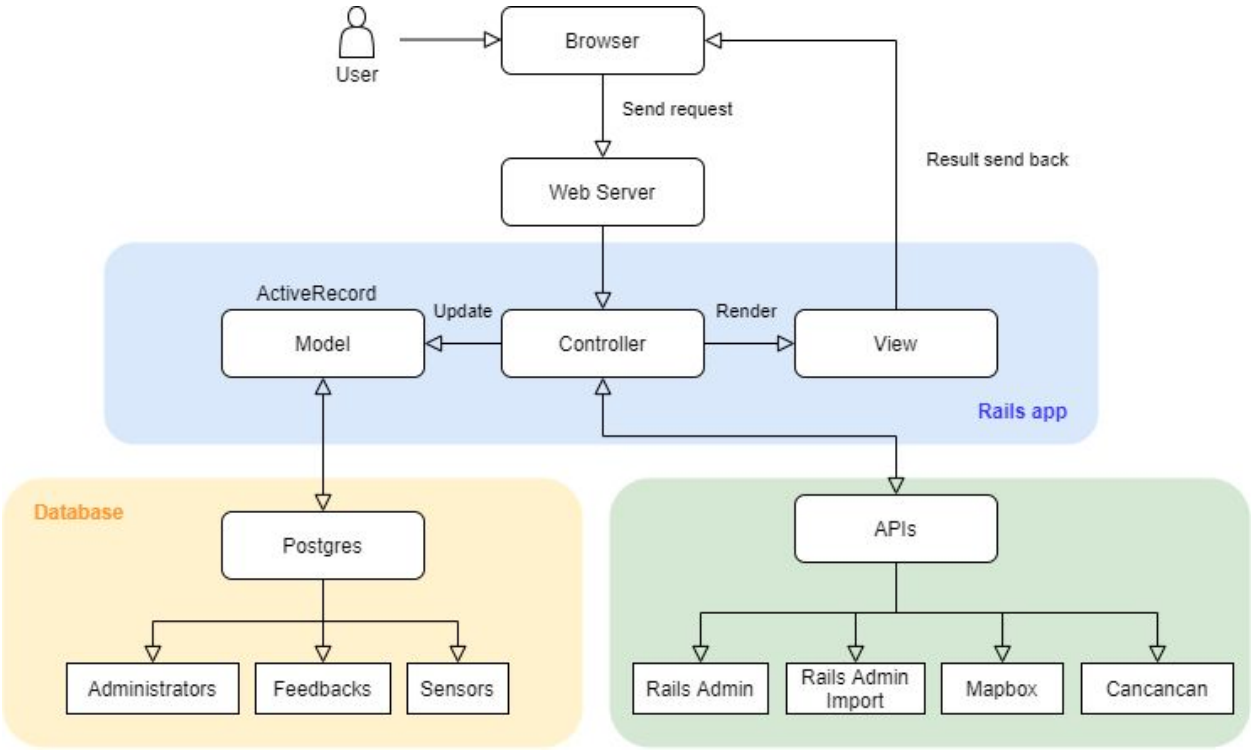
3 Architecture

3.1 Architecture overview

Figure 1 shows the complete architecture of our app. To elaborate, we used ***Ruby on Rails***[1] (RoR) as our app framework, which utilizes a model-view-controller pattern that separates the app work into three subsystems, and each subsystem is responsible for a different set of actions. In the RoR framework, when a user sends a request to the app server, the controllers will receive the request for the application and then works with the model to prepare any data needed by the view, and the controllers will also interact with external APIs like Mapbox to complete certain requests. The technical details of these APIs will be explained in the following sections. The database is implemented with ***PostgreSQL***[2], an open-source relational database management system which has native support for using SSL connections to encrypt client/server communication, thus ensuring data security. Our app is deployed on ***Heroku***[3], a cloud service provider which offers free to low-cost infrastructure setup and convenient add-ons. It allows the developer to focus on code instead of infrastructure and it enables scaling both horizontally and vertically in case the app needs more performance.

---

[1] "Ruby on Rails | A web-application framework that includes ...." https://rubyonrails.org/. Accessed 7 May. 2020.
[2] "PostgreSQL - Wikipedia." https://en.wikipedia.org/wiki/PostgreSQL. Accessed 7 May. 2020.
[3] "Heroku." https://www.heroku.com/. Accessed 12 May. 2020.



3.2 Database and Schema

PostgreSQL is the most popular and advanced open source object-relational database, which is considered superior to others when it comes to out-of-the-box security. The database security for PostgreSQL is addressed at several levels[4]:

1. All files stored within the database are protected from reading by any account other than the Postgres superuser account.
2. Connections from a client to the database server are, by default, allowed only via a local Unix socket, not via TCP/IP sockets.
3. Client connections can be restricted by IP address and/or user name.
4. Each user in Postgres is assigned a username and (optionally) a password. By default, users do not have write access to databases they did not create.
5. Users may be assigned to *groups*, and table access may be restricted based on group privileges.

In our case, data is securely stored in the database and is only accessible by authorized administrators via the local Unix socket. There are three tables in the PostgreSQL database which are responsible for storing information about the administrators, users feedback and sensors (table 1-3).

[4] "Documentation: 7.0: Security - PostgreSQL." https://www.postgresql.org/docs/7.0/security.htm. Accessed 7 May. 2020.

In the administrator table (Table 1), name is the name of the admin. Email is the unique email for admin login and activation receival. Access code is the required secret code during account creation and only authorized personnel with correct access code can register for an admin account. Admin's raw password is encrypted and securely stored in the database, and user validation is done by computing its password digest and comparing the result to password_digest stored in the database.

	name	email	access_code	password_digest
Data type	string	string	string	string
Example	John Smith	js@gmail.com	pororo	$2a$12$WgjER5o

Table 1. Schema for the Administrator table with trivial columns omitted.

For the Feedback table, the issue is the content that the user reports to the backend. Once a user submits a feedback for a sensor, its sensor id will automatically be filled in by the system under the topic column. Therefore, issues and sensors are associated with each other and the admin can search for a sensor id in the admin's page to list all the issues submitted by users for that sensor.

	issue	topic
Data type	string	string
Example	Camera is facing my window	sensor 2319

Table 2. Schema for the Feedback table.

For the Sensor table, sensor_type is the type of sensor in the database, which for now includes antenna, air quality, camera, counter and other. Owner indicates who owns the sensor, i.e., City of Boston or 3rd party vendors. Description is an optional text to describe the sensor. Location is the full address where the sensor is located. Longitude and latitude are the geographic coordinates of the sensor. The hidden field decides if the sensor will show up in the app and is set to false by default. This is especially useful when some sensor is under maintenance but we still want to keep its information in the database.

	sensor_type	owner	description	location	longitude	latitude	hidden
Data type	string	string	string	string	float	foat	boolean
Example	camera	Verizon	...	2 Oak st	-71.0657	42.34867	false

Table 3. Schema for the Sensor table with trivial columns omitted.

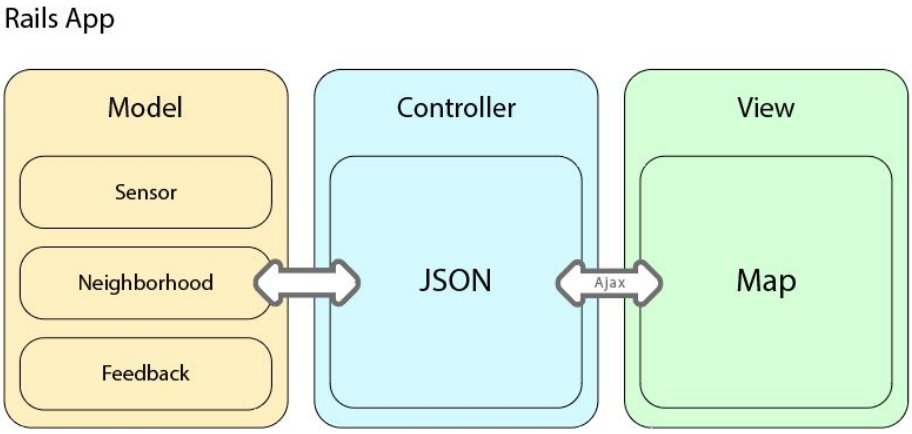
3.3 APIs

3.3.1 Mapbox

Mapbox[5] is an open source mapping platform for custom designed apps and is the key component in our application architecture. This JavaScript API is the building blocks to integrate location into mobile or web applications. When Mapbox JavaScript library using WebGL renders maps from custom selected vector tiles and Mapbox styles into the Rails view, it uses JSON files created from the Rails controller to display the sensor in PostgreSQL database (which is derived from seed file or information the administrator has input via ***Rails Admin***). Each object displayed on the map uses the GeoJSON format (an open standard format designed for representing simple geographical features along with their non-spatial attributes) to contain its data about longitude and latitude, along with client side information including sensor type, sensor label, sensor ownership, and sensor description. To match this GeoJSON format that Mapbox API requires, the structure of the JSON file is manipulated using ***jBuilder***[6] for each object. When the map loads in the Rails View, the dots representing each sensor reads these location information from the JSON file and accordingly displays them in the correct place. When users of the application click on each sensor, or use a search function enabled by the mapbox's ***geocoder*** functionality, the Rails view communicates with the Rails controller via Ajax to access the data linked to each sensor. Note that geocoding using Mapbox geocoder is only targeted for the client side, which means that the geographical coordinates are provided from the name of the location when users search for that location (e.g. searching "Boston City Hall" returns [-71.057991, 42.360405]). When dots are clicked, the "properties" section of the GeoJSON is accessed to look for the id of the corresponding sensor, and this id is sent to Rails Controller to display the appropriate information on the information container in the Rails View. Similarly, when the users submit feedback using the feedback textarea placed on the information box, the id of the clicked or searched sensor is used as a parameter for the controller action to display the inputted id with the feedback so that the administrators can identify which sensor the feedback is about.

```
{
    "type": "FeatureCollection",
    "features": [
        "type":"Feature",
        "geometry": {
            "type":"Point",
            "coordinates":[
                -71.050054,
                42.360152
            ]
        },
        "properties": {
            "id":186,
            "sensor_type":"camera",
            "ownership":null,
            "description":"surveillance camera"
        }
}
```

[5] "Mapbox." https://www.mapbox.com/. Accessed 14 May. 2020.
[6] "JBuilder - Wikipedia." https://en.wikipedia.org/wiki/JBuilder. Accessed 14 May. 2020.



Other than simply providing the data for the JSON file, more active communication between the model and the controller occurs when the application does “server side geocoding” with ***Rails Geocoder***[7] gem. For the convenience of the app administrators, the application is built to take street addresses instead of raw geographical coordinates. When street addresses are entered into the location column, they are automatically converted into coordinates upon saving, using the actions provided by the geocoder gem. ***Reverse Geocoding*** is also possible, in case the administrators already have the coordinates data. In this case, the location column is populated using the information derived from the latitude and longitude columns. The location, latitude, and longitude information thus provided is used in the map to display the sensors’ address in the information box, and configure the sensor location respectively.

3.3.2 Rails Admin

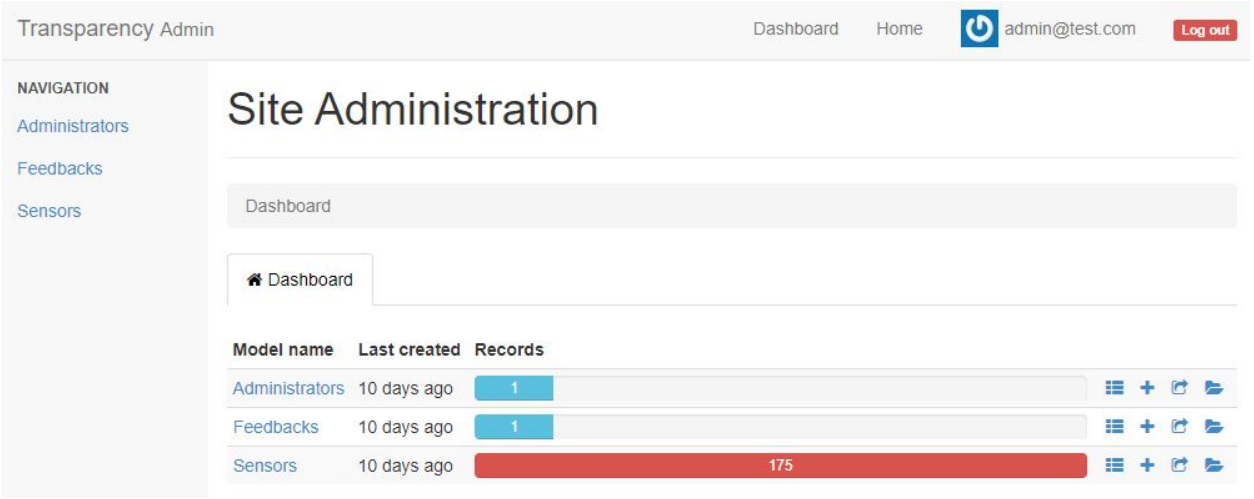
To provide administrators a better user experience in data managing, we used ***Rails Admin***[8], a rails engine that provides an easy-to-use interface, for site administration. It allows administrator to:

- CRUD any data with ease
- Search and filtering data
- Export data to CSV/JSON/XML format
- Import data from CSV/JSON/XML format (via RailsAdminImport add-on)
- Automatic form validation
- Authorization
- Authentication (via CanCanCan)

The administrator site is password protected and only authorized personnel is allowed to view and manage the data. In the site administration page, a summary of all records in the

[7] "alexreisner/geocoder: Complete Ruby geocoding ... - GitHub." https://github.com/alexreisner/geocoder. Accessed 14 May. 2020.
[8] "sferik/rails_admin: RailsAdmin is a Rails engine that ... - GitHub." https://github.com/sferik/rails_admin. Accessed 14 May. 2020.

database is displayed, and admins are able to manage specific sets of records in the corresponding pages. The detailed usage of the admin interface is explained in the admin guide.



3.3.3 Rails Admin Import

Rails Admin Import[9] provides a plugin functionality to add generic import the Rails Admin from CSV, JSON and XLSX files. Model instances can be both created and updated from import data. Any fields can be imported as long as they are allowed by the model's configuration. Associated records can be looked up for both singular and plural relationships. In the Rails Admin Import gem, both updating existing records and associating records requires the use of mapping keys. In our case, to uniquely identify each sensor, we used both longitude and latitude as mapping keys, and sensors with identical longitude and latitude data will be updated during record importing. This avoids accidental duplicated importing and ensures data authenticity.

3.3.4 CanCanCan

CanCanCan[10] library is used to authorize access to the site administration page for the administrators. In our configuration, non-logged in users are not allowed to access the site administration page and will be redirected to the login page even if they manually typed in the url. In the future, access to certain pages within the administration site can be restricted only to super admin to ensure better data security. Other admin roles can also be defined with read-only access or partial permissions tailored for actual application.

---

[9] "stephskardal/rails_admin_import: Rails Admin Import ... - GitHub." https://github.com/stephskardal/rails_admin_import. Accessed 14 May. 2020.
[10] "CanCanCommunity/cancancan: The ...." https://github.com/CanCanCommunity/cancancan. Accessed 14 May. 2020.

4. Status and Future

Upon presenting the application to Nayeli and a few of her colleges that also work in the Boston government, the application received a very positive response. Many of the questions concerned the future of the application: possible additional functions that we could implement, and how the application would/could continue after the semester ends. Because of the current coronavirus situation, it would be impossible to implement this app into Boston and get it into the hands of citizens at this moment. However, the Boston team did seem to be extremely interested in deploying this application into the real world eventually, once there was an opportunity.

We have many ideas on what we'd do if given more time on this application. As we were working, there were many issues that we ended up solving and some that we knew had to be solved/addressed if we were given more time and energy. Below we've compiled a list of specific features and measures we would like to add into the application in the future.

- Loader.io scalability test
- RabbitMQ for queueing when bulk importing sensors to database
- Database tuning and indexing
- Redis for database caching during map rendering
- Heroku scheduler to daily check database records (maybe move geocoding & reverse geocoding there)
- Change monolithic app structure to Service Oriented Architecture (SOA)
- Implement load balancer to send traffic to different server and separate request for static and dynamic content
- Action Mailer to send super admin email notification about database change/new comments and other relevant information
- Add map functionality that allows a user to see first person view, which would require more specific information about the sensors, such as above ground/underground/height (requested by CoB)

5. Team

Even though our team this semester was truly a dream team in every sense, this app could continue after this semester, even though 75% of our team will graduate. Belle will be around for the next year or two to fix all the bugs and add all the features the client desires. Finding more team members would also be possible, as there was further interest in this project (though they would in no way replace the team we had this year).

6. Resources

Github
Gitbook
Admin guide