Bus Tracker: Final Report

Ian Will, Jason Cluck

Contents

List of Figures

Introduction and Motivation

Public transportation in the District of Columbia is critical to many of its residents and employees. The District's public transporation is managed by the Washington Metro Area Transit Authority (WMATA). WMATA maintains a system of rail and buses that serve most location in the district. The WMATA rail system is simpler than the bus network and has many maps that are useful for discovering which rail lines server particular regions. The bus system serves more locations that rail, but can be significantly more perplexing for a new rider. There are a large number of routes and no comprehensive map, and buses rarely adhere to posted schedules.

1.1 Problem Statement

The Bus Tracker web application provides an interactive map-based overview of the WMATA bus system. It uses a web API provided by WMATA to query current bus locations, predictions of arrival times at stops, and full listings of route and stop locations. This information is then displayed in a clear and interactive manner that displays high-level overviews with easy drill-down for pertinent details (as encouraged by Edward Tufte [1]).

Requirements Analysis

2.1 Use Cases

The application is intended for use by those considering or planning to travel on the WMATA bus system. The primary use case is a smart phone user considering how to catch a bus to an intended destination. Someone in that situation wonders how long they will be waiting for a bus to arrive at the nearest stop. They may like to consider alternatives to the bus system such as rail, taxi, or alternate bus route depending on how long they will need to wait for a bus. They may also like to use Google's traffic overlay to estimate how long the bus will be delayed.

The application will help by showing the anticipated time until the next bus arrives, showing where the next arriving bus is going, and showing nearby stops and routes that may be equally suitable for the intended destination. This person will probably make location information available to the application through their browser. The web application may use this information to filter routes and buses that are display to improve performance. It may also use the location to suggest alternate routes with more promptly ariving buses.

A second use case is someone at their home or desk computer considering when to leave to minimize wait time at a stop without missing a bus. The concerns in this case will be similar to to the concerns of the smartphone user, but location data may not be available. The person may want a reminder when they need to leave in order to catch their bus.

2.2 Threshold Requirements

Threshold requirements describe the minimal set of functionality for a usable product. These are the requirements that we plan to accomplish.

1. Display a map of the DC metro region in a web browser
2. Display the browser's reported location on the map if current location is available from the browser
3. Display all WMATA bus routes on the map
4. Display the current locations of all WMATA buses on the map
5. Visually indicate bus direction along with bus position displays
6. Display bus routes using different colors to visually differentiate routes
7. Display bus positions using the same color used for its route
8. Update bus position markers with current positions at five second intervals
9. Display bus stop locations on the map
10. Display the following detail information about a bus stop by clicking on the stop marker
    (a) Display the anticipated wait time for the next bus for each route that intersects a given stop
11. Display the following detail information about a bus by clicking on the bus marker
12. Display the stop's WMATA id number
13. Display all bus routes that serve the stop
    (a) Bus route name (e.g. "5A")
    (b) Bus headsign (e.g. "DULLES AIRPORT")
    (c) Bus direction (e.g. "WEST")
    (d) Positional uncertainty circle around the bus marker

2.3 Objective Requirements

Objective requirements describe functionality that would improve the quality or usefulness of the application, but are not strictly necessary for a useful product. These are things we would like to accomplish, but are unsure if time will allow.

1. Draw DC Circulator bus positions and routes (DC Circulator is not part of the WMATA system)
2. Provide filtering to limit the displayed buses and routes
   (a) Filter by only showing routes, stops, and buses matching a route name
   (b) Filter by only showing routes, stops, and buses matching a headsign
   (c) Filter by only showing routes with stops wihtin some proximity to the device's current location
   (d) Filter by only showing routes with stops wihtin some proximity to an arbitrary location
   (e) Combine filters using boolean logic, for example only showing routes with stops near current location and some arbitrary location
3. Send a text message or email reminder when a bus is some number of minutes away from a stop

Design

The project to create a live map of DC's Washington Metro Area Transit Authority (WMATA) bus system will be built using the Ruby on Rails web framework, the Google Maps API, WMATA's Transparent Data Sets API, and Twitter's Bootstrap CSS API.

The Google Maps API is used to display the background map, route paths, and bus marker overlays. Figure 3.1 shows an overview for how these APIs connect to the Rails web server we're building.

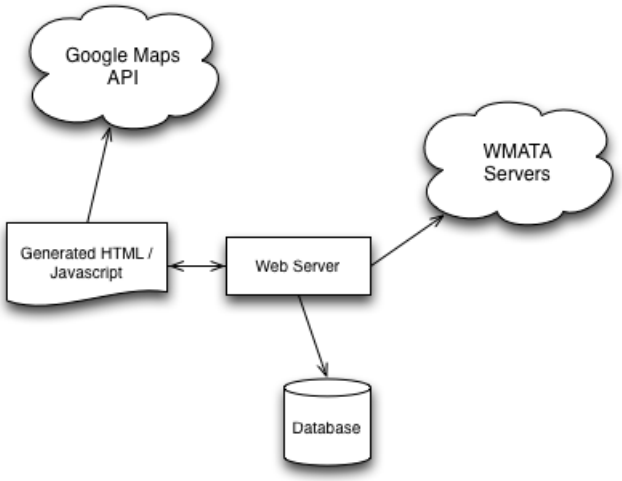

Figure 3.1: API integration overview

The Rails design philosophy favors adopting standard conventions for naming, folder structure, and design patterns across all Rails applications. This accelerates web development, saving a lot of configuration and rudimentary structuring and naming decision when starting a project. It also allows

the framework to provide numerous helper tools that automate as much as possible.

3.1 Model-View-Controller

The fundamental design pattern adopted by Rails is Model-View-Controller (MVC). The premise behind MVC is to separate user-interface code (the View), data structures (the Model), and the business logic that connects them (the Controller) into separate modules that can be developed, unit tested, and maintained in isolation. Figure 3.2 shows the MVC separation as used in the Bus Tracker application.

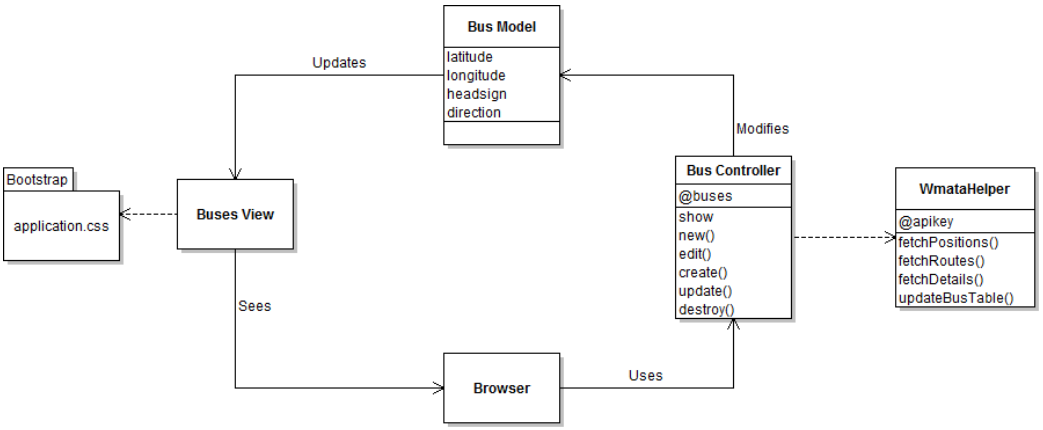

Figure 3.2: MVC control pattern

When an HTTP request comes to the HTTP Server, it first goes to the rails router, which determines which Controller should handle that request. The controller then uses the Model classes to get the current state of data (e.g. the current position of buses from the database), and passes that information to the View classes. Rails stores the View logic as HTML with additional preprocessor directives for special scripts called "partials" including Embedded Ruby and CoffeeScript. When these partials are processed by the appropriate preprocessor engine, they have access to variables and data structures provided by the controller. The preprocessor engine dynamically generates standard HTML and Javascript based on the partial scripts and the data provided by the controller. These conventions and tools make it

easy to prevent business logic from creeping into view code. They provide a standard idiom for passing data between the model, controller, and view.

Rails adopts the Representational State Transfer (REST) architecture for the HTTP interface. It uses the inherent HTTP methods (GET, POST, PUT, DELETE) to execute the corresponding Create, Read, Update, Destroy (CRUD) operations on the data model. This approach is contrasted by the alternative (non-REST) approach of relying predominantly on the HTTP GET method and controlling CRUD behavior using special parameters embedded in the URI [2]. Our Bus Tracker application doesn't provide a user-facing capability to create, update, or destroy the model since model updates are fetched from WMATA servers rather than driven by user input. Therefore the POST, PUT, and DELETE routes have no action associated with them in the controller.

The GET requests retrieve different types of data depending on the URI requested. Requesting the root (GET / or GET /index.html) provides the overview map display. Requesting GET /buses or GET /buses.json returns the current state of the buses model. Request GET /stops requests data purely on the stops. These likely won't be used directly by users, but they are used in the Javascript that asynchronously updates the map display. Periodically a Javascript request is made to /buses.json to retrieve an updated array of bus positions in the JSON format.

Some of the API features specific to Bus Tracker are listed in Figure 3.2. The controller relies on fetching data from the WMATA servers which are accessed via a helper class - WmataHelper. WmataHelper fetches the current position of all buses from the WMATA servers and stores that data in using the model classes.

Bus Tracker also uses is Twitter's Bootstrap API which provides additional CSS options. Combining Bootstrap with the View can create dynamic web pages that can render correctly on phones or desktop computers with minimal overhead.

3.2 Model Relationships

Bus Tracker will contain information regarding all the buses served by WMATA. This includes bus information (including positions), stop information, and route information (including way points describing the route). Figure 3.3 shows the entities and relationships between the WMATA data used by Bus

Tracker. This is the information that will be stored locally after retrieving the information from WMATA.

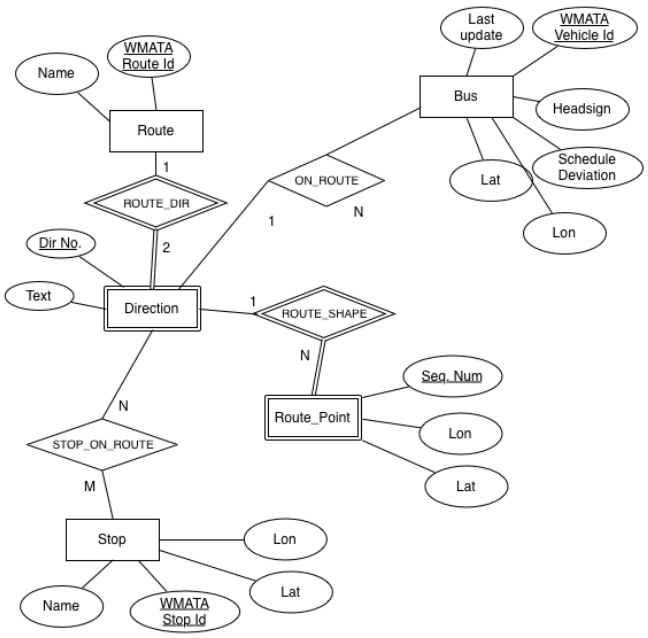

Figure 3.3: Model relationship diagram

Figure 3.4 shows the schema that Bus Tracker uses to store information about the current state of the WMATA bus system. This persists all necessary information on routes, buses, and stops. Underlined fields indicate primary keys, arrows indicate foreign key dependencies on primary keys from other tables.

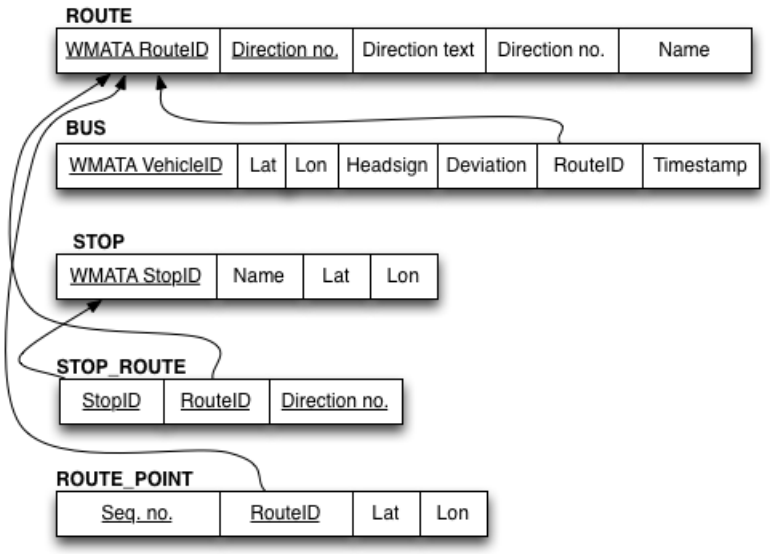

Figure 3.4: Schema for WMATA API

Implementation Details

Bus Tracker was implemented using the Ruby on Rails web framework, the Google Maps Javascript API, and asynchronous Javascript and XML (AJAX). The Rails application interfaces with WMATA servers and maintains its own representation of the current state of the WMATA server in its databases. It polls this periodically and updates its database accordingly.

The WMATA system's information is mirrored in our own database for two reasons. One motivation is that the WMATA API servers respond slowly. Retrieving bus position updates for the entire system takes about 10 seconds to complete. Our own implementation can respond in about one second when running from a SQLite database on the local machine, and would potentially be faster with a more powerful database. Thus mirroring the data allows faster responses to front-end AJAX calls. A second motivation is that the WMATA API is subject to usage restrictions of a few calls over a few seconds. If the web application is to scale to support multiple concurrent users, those usage restrictions will be violated. Mirroring the system information in our own database allows more flexibility to scale up to more significant user loads.

The rails server implements a traditional Create / Retrieve / Update / Destroy (CRUD) interface when the paths routes/, stops/ and buses/ are visited. The main page, however is is a full page map with icons that are updated using (AJAX) (shown in Figure A.1). The asyncronous javascript calls interface with the Rails server to retrieve system state using Javascript Object Notation (JSON). These JSON representations are parsed in the client-side javascript and the map icons are updated accordingly.

Information bubbles appear when icons are clicked (shown in Figures A.2 and A.3). These are populated with HTML generated by the Rails server. They retrieve HTML from the corresponding REST style url e.g. /buses/1 for the content of the information bubble for a bus with an id of one. Similarly stop information windows are populated from the path

/stops/1. Next-bus predictions for a particular stop are retrieved using the path /stops/1/prediction. These paths can be entered directly into the browser, and the information will be displayed outside of the map context.

4.1 Google Maps

The system processes significant quantities of data. It supports over 11,000 stops in the WMATA system, over 350 routes, and almost 650 buses. Each route is described by a sequence of latitude, longitude points. The complete path description of all routes in the system is 15 MB when represented as KML. The initial approach rendered all routes in javascript using google.maps.Polyline from the Google Maps Javascript API. However, the rendering process during the initial page load was slow. Routes would slowly appear on the map one by one over about 15 or 20 seconds. To speed the route rendering process, we pre-proccessed the routes into KML and used google.maps.KmlLayer to render each. This improved rendering somewhat, though loading and panning in Bus Tracker is still noticably slower with routes.

Google Maps renders KML in one of two ways: small KML files are rendered on the client side; but large KML files are loaded, tiled, and cached on Google's servers. The Google Maps API also imposes size limits on each KML layer [3], 3 MB maximum size of KMZ with 10 MB maximum size uncompressed. To accomodate this, the 15 MB of raw KML was split into four regions–Virginia, DC, South Eastern Maryland and North Western Maryland. Each of these meet the Google size limitations. Google also requires that these files be available at a URL that its servers can reach. When running a development server on localhost, Google can not access local resources. Therefore the four KMZ files are placed on a public web server during development to facilitate running from localhost. When deployed, the files would be hosted on the same public web server that hosts Rails.

Bus and stop icons are displayed using Scalable Vector Graphics (SVG). This is by far the most flexible method of using custom icons with Google Maps. Icons specified with SVG can be scaled, rotated, and dynamically colored. The alternative is to use PNGs, which offer less flexibility.

4.2 WMATA API

We were dissapointed at the update rates supported by the WMATA API. New bus positions were only available every two to five minutes. The initial vision for the project was that the accurace placement of bus icons on the map would be sufficient for bus riders to determine the wait (or whether they'd missed the bus). A delay of five or more minutes is significant when your walk to the stop takes five or ten minutes. Someone might delay leaving the house thinking they had more time, and then miss the bus because the data that drove their plans was out of date.

Significant effort was invested in insuring it was WMATA and not our own code that introduced the delay. This effort spanned a few different designs for retrieving data from WMATA. The first approach was to poll the server every time our server received a request, wait for a response from WMATA, and return that to the querying AJAX process. This worked, but introduced delays of about 10 seconds for the AJAX calls. A second approach used threads inside the Rails server to for the WMATA polling. When the Rails server received a request, it answered immediately, but queued a request to WMATA. This worked for a while, but didn't mesh well with the Rails architecture. In Rails, each controller (which processes an incoming request) is instantiated per request. A mutex can be stored at the class level, but classes can even be dynamically reloaded (a convenience for development), which reset the mutex and results in concurrent modification problems. The third approach moves the WMATA polling outside of the Rails server and into a separate process at the operating system level. This process could be configured as a cron job which recurrs at 10 second intervals, or using a more sophisticated approach integrated with Rails (such as the foreman, or thin gems). This third approach seems to work the best. It prevents multi-threaded access to the database through the Rails ActiveRecord classes which are not thread safe. It also allows the interval of WMATA polling to be separate from the interval used in the front-end Javascript, so icons can be updated more frequently (even if their position hasn't changed).

4.3 Bootstrap

Twitter's Bootstrap API was used to overwrite the default CSS provided by Rails scaffolding. Bootstrap has recently emerged as a versatile toolbox of

html and css classes. Bootstrap was used in Bus Tracker to include a collapsible navigation bar that allows for the application to be used on browsers with smaller windows such as mobile devices. The collapsed navbar provides a list of buttons that can be recessed if the user wants to see more of the map, making it ideal for this application. Other uses include a Modal unit for the About section which provides a nice javascript enabled window which drops down over the UI. Utilizing Bootstrap did create some problems with Google Maps but this was resolved by customizing the CSS classes. [4]

4.4 Deployment on Heroku

Heroku was chosen as the hosting for the production environment due to its easy of deployment with Rails applications and usage with Git. Although deploying with Heroku was easy at first, there were some things that required work additional work. The first problem we encountered was how to continually poll for data in the background while not stalling the main application thread. This was done by using a Heroku Worker in conjunction with the Delayed Job gem. This gem allows any Active Record object to be added to a priority queue which the Worker handles. With this solved, there still was the issue of having the code run every 10 seconds (the minimum value before exceeding the threshold). To fix this, A Heroku Clock Singleton process was added (in the form of an additional Dyno) that kept track of what tasks were ready to be run. The Clockwork gem combines with the Heroku Clock to specify what tasks the Heroku Clock should run. An example of a poll would include the Heroku Clock seeing that it needs to run something (as specified by Clockwork), and then adds it to the Delayed Job queue. The Heroku Worker picks up the task from the queue and processes it – updating all of the bus locations. [5]

Test Cases

5.1 Test Overview Page

Steps

1. Run the Rails server rails server
2. Run the update daemon script/daemon run poll_buses.rb
3. Load root web page (http://localhost:5000)

Verification Criteria

1. Route KML should load, showing bus routes with various colors
2. Bus icons should display after a brief delay
3. At least a few bus icons should be somewhat opaque
4. Bus colors should match route colors
5. After a few minutes, at least a few buses should bounce to indicate new position data

5.2 Test Bus Information Window

Steps

1. Run the Rails server rails server
2. Run the update daemon script/daemon run poll_buses.rb

3. Load root web page (http://localhost:5000)
4. Click on a bus icon

Verification Criteria

1. Verify that the information bubble appears
2. Verify that the window is attached to the bus
3. Verify that the window contains destination
4. Verify that the window contains age of information
5. Verify that the window contains vehicle id number
6. Verify that the window contains direction
7. Verify that the window contains deviation from schedule (minutes late or early)

5.3 Test Route Focus

Steps

1. Run the Rails server rails server
2. Run the update daemon script/daemon run poll_buses.rb
3. Load root web page (http://localhost:5000)
4. Click on a bus icon
5. Click the "Show Only D2" button in the info window

Verification Criteria

1. Verify that only the selected route is displayed on the map
2. Verify that the route's stops are displayed
3. Verify that the route's buses are still displayed

4. Wait for bus position updates, verify that they occur within about 2 to 5 minutes

5.4 Test Stop Information Window

Steps

1. Run the Rails server rails server
2. Run the update daemon script/daemon run poll_buses.rb
3. Load root web page (http://localhost:5000)
4. Click on a bus icon
5. Click the "Show Only D2" button in the info window
6. Click on one of the stops

Verification Criteria

1. Verify that the stop window is displayed
2. Verify that after a few seconds, the next bus prediction table appears

5.5 Test Non-Map Interface

Steps

1. Run the Rails server rails server
2. Run the update daemon script/daemon run poll_buses.rb
3. Load routes page in a browser (http://localhost:5000/routes/)
4. Load stops page in a browser (http://localhost:5000/stops/)
5. Load buses page in a browser (http://localhost:5000/buses/)

Verification Criteria

1. Each page should show a table listing the elements in the system

Task Breakdown and Timeline

Tasks to be done by: November 10, 2012

1. Initial rails setup [Cluck]
2. CSS styling using twitter bootstrap [Cluck]
3. Display map on index page [Cluck]
4. Fetch bus positions from WMATA servers [Will]
5. Store WMATA response in database [Will]
6. Draw markers on map for bus locations [Will]
7. Draw lines on map for bus routes [Will]
8. Display initial map area based on geo-location reading [Cluck]
9. Show info window when bus marker is clicked, including the following [Will]
   (a) Route name
   (b) Schedule deviation (lateness)
   (c) Time of last position update (data staleness)
   (d) Headsign
   (e) Direction

Tasks to be done by: November 20, 2012

1. Fetch routes from WMATA servers [Will]
2. Fetch stops from WMATA servers [Will]
3. Store routes in database [Cluck]
4. Store stops in database [Cluck]

Tasks to be done by: December 3, 2012

1. Display bus stops with markers on map [Cluck]
2. Control WMATA polling to avoid exceeding usage limits [Will]
3. Show info window when stop marker is clicked [Will]
   (a) Show which routes and directions the use the stop
   (b) Show next bus wait time projections for each route that uses the stop
4. Show info window when stop marker is clicked [Will]
   (a) Retrieve HTML from Rails stops/:id path
   (b) Use AJAX to update next bus arrival predictions from WMATA
5. Orient bus markers according to direction traveled [Will]

Bibliography

[1] E. Tufte, *The Visual Display of Quantitative Information*, Second. Graphics Press LLC, 2006.

[2] *Ruby on rails*. [Online]. Available: `http://rubyonrails.org`.

[3] (Dec. 2012). Kml support - keyhole markup language - google developers, [Online]. Available: `https://developers.google.com/kml/documentation/mapsSupport`.

[4] *Twitter bootstrap*. [Online]. Available: `http://twitter.github.com/bootstrap/`.

[5] *Heroku*. [Online]. Available: `http://www.heroku.com`.

User Manual

The Bus Tracker web application immediately presents an overview of the current state of the WMATA system. All routes are shown as KML overlays, and current positions of buses on those routes are indicated with bus icons. This initial state is shown in Figure A.1.

Bus icons move as their position information is updated on WMATA's servers. These updates do not come from WMATA as frequently as desired, which is a limitation of their system. Buses typically are updated within a two minute time span, but the delay can sometimes be much longer. When position updates are received for a bus, its icon bounces a few times to indicate a change in position.

The age of a bus position can be quickly determined based on the opacity of a bus icon. A transparent icon has position information that is more than five minutes old. The more opaque a bus icon is shaded, the more recent its position information. The exact age of the bus position information can be seen in the bus information window that displays when clicking on a bus icon. An example is shown in Figure A.2. It includes route name, direction, headsign, age of position data, vehicle id number, and deviation from the posted schedule.

Initially all buses and all routes are displayed. This can be a nice overview, but can also be visually overwhelming. Bus Tracker allows you to focus in on a route, showing only its path, buses, and also displaying the stops thoses buses serve. There are two ways to focus on a route. One is by clicking on the route path, then clicking "Show Only Route D2" on the tool bar (where D2 is route clicked). The second way to focus on a specific route is to click the "Show Only D2" button in a bus information window. This will focus on the route that the bus is serving.

When a specific route is in focus, the stops on that route are also shown. Clicking on a stop icon brings up a stop information window (Figure A.3).

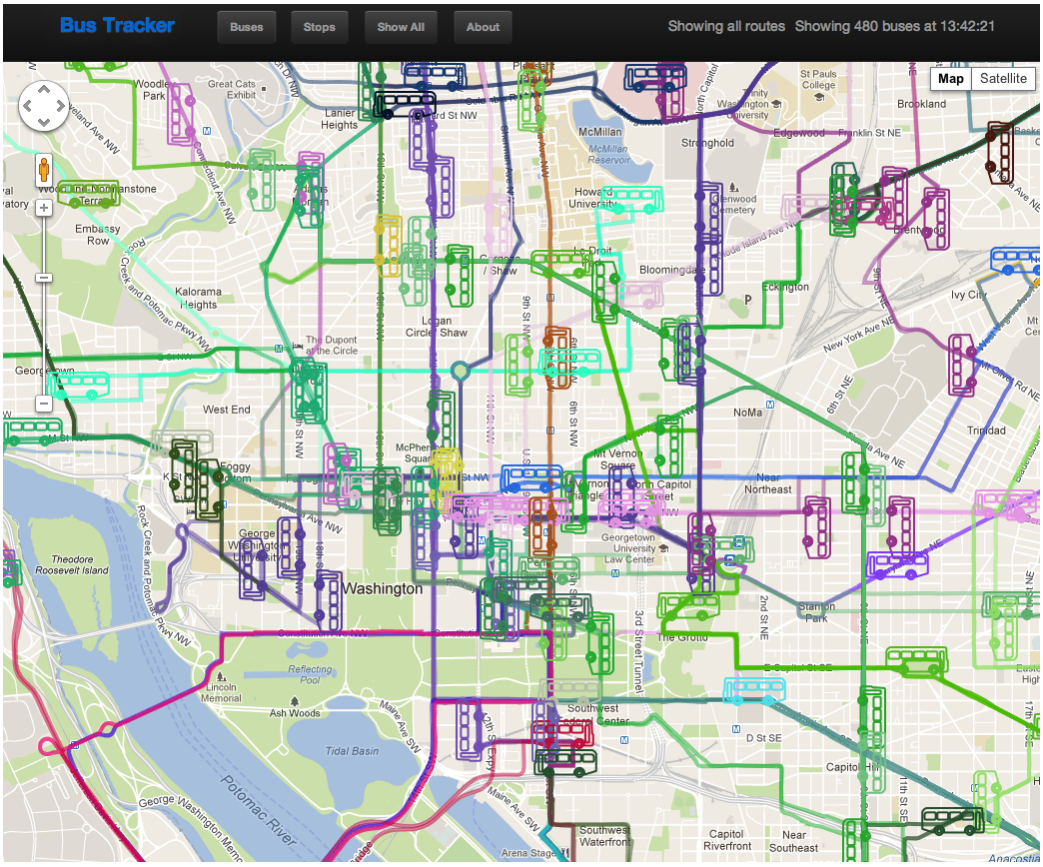

Figure A.1: Overview of WMATA Bus System

Stop information windows show the stop's name, its WMATA stop ID (which other applications use to look up bus arrival predictions), a table of predicted bus arrival times, all routes that serve this stop, and the stop coordinates.

The bus arrival predictions shown in the stop information window are more accurate than bus position data. They are provided by WMATA's API, and project bus arrival based on last-known location and average bus speed.

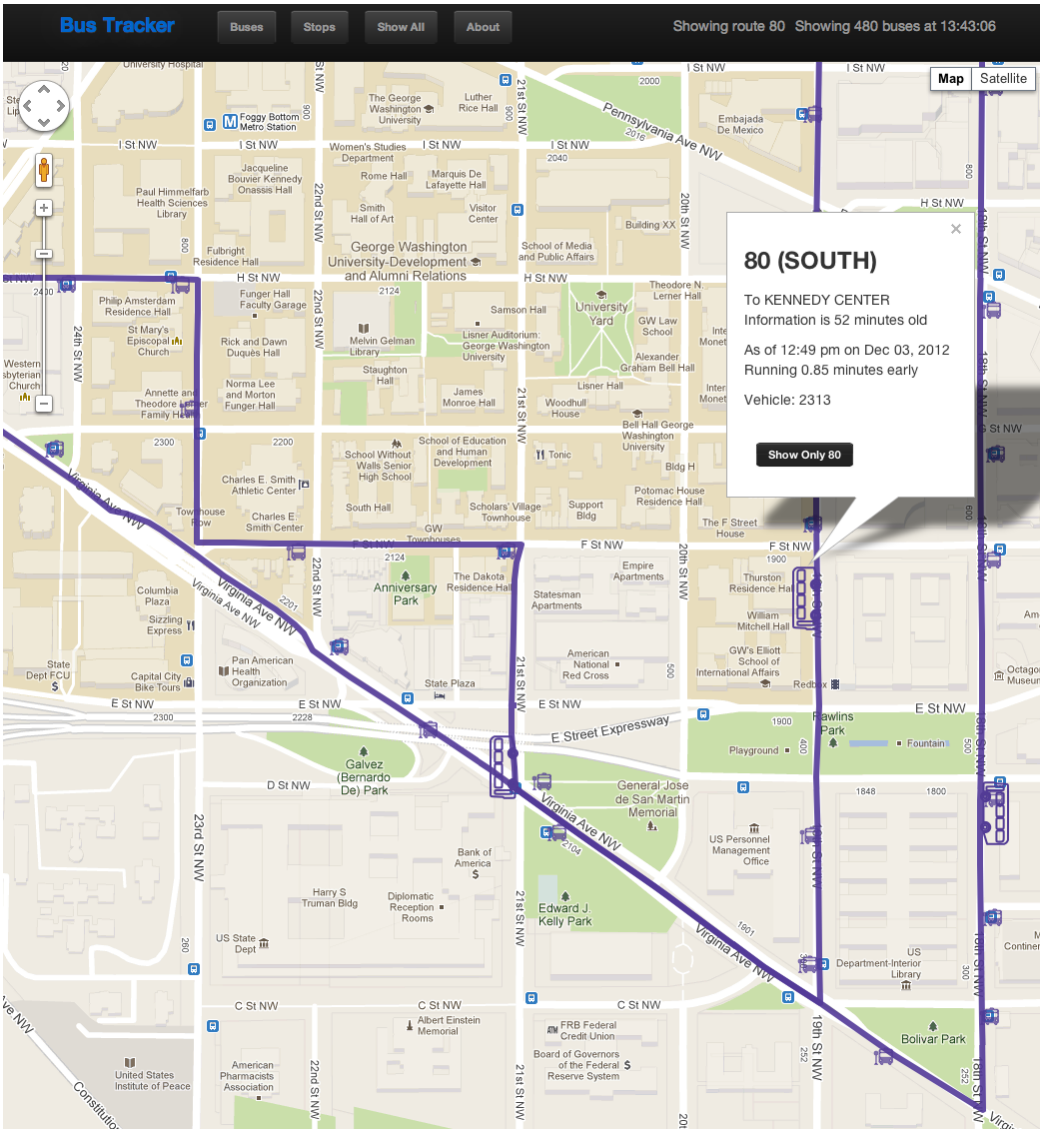

Figure A.2: Bus Detail Window

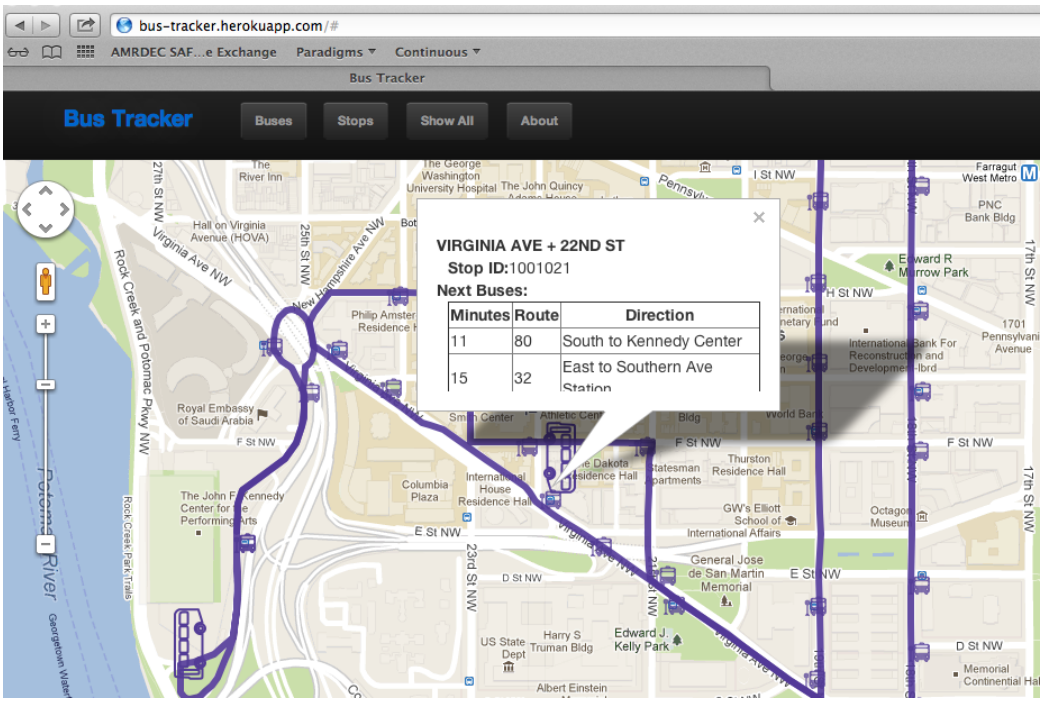

Figure A.3: Stop Detail Window