Live Map of the DC Bus System: Design

Ian Will, Jason Cluck

Introduction

Public transportation in the District of Columbia is very important to most of it's residents and employees. This project aims to provide users with recent bus data to ensure the most productive transportation. The project will manifest as a web application named Bus Tracker that can be accessed through any device capable of supporting a web browser. Some of the useful data that will be provided through this application will be the buses' current location, their expected arrival times, and their heading. Using this information will allow users to make informed decisions in real-time instead of relying on applications that simply provide arrival times of the buses.

Design

The project to create a live map of the DC Bus System will be built upon preexisting frameworks in order to create a functional system while shortening the development process. The project is built upon the Rails framework which favors convention over configuration when creating web-based applications. One example of how Rails alleviates some architectural decisions is it's use of Model-View-Controller (MVC).

Model-View-Controller

MVC is a popular software architecture that has been gaining popularity due to it's advantages for developing web applications. It is also the default architecture choice for rails which reduces total implementation time. The basic reason for using MVC is that is separates the representation of the

data from actual user interaction with the data. The MVC for Bus Tracker is shown below in Figure 1.

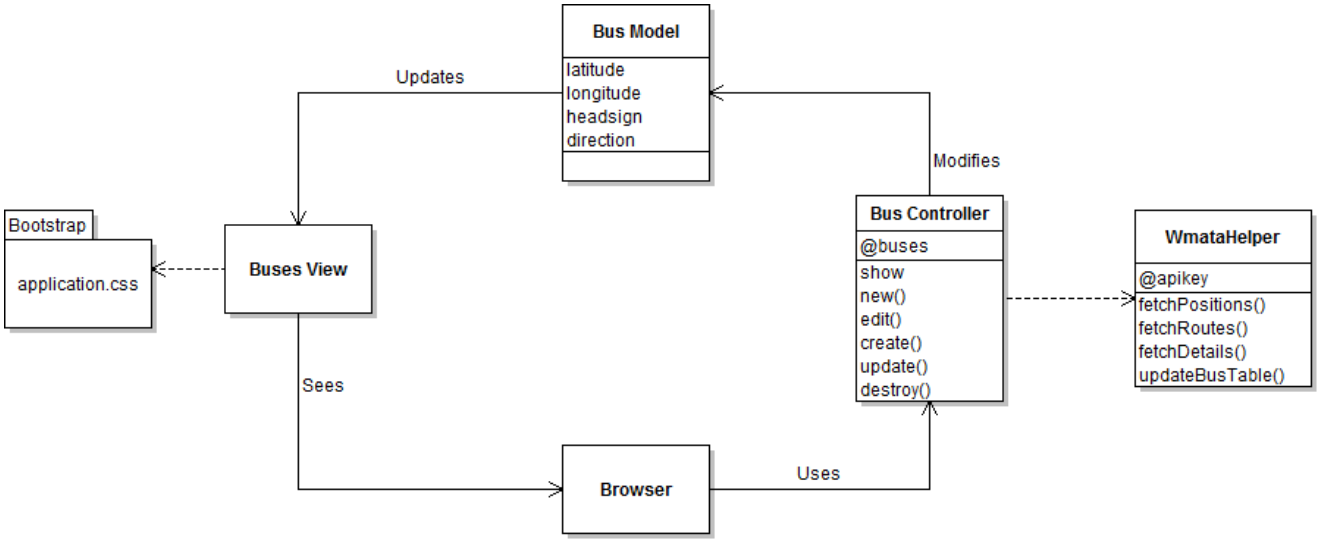

Figure 1: MVC control pattern

This MVC handles user requests in the following way. If a request is sensed via the browser (from the user or an update timer), the controller is called and told to modify the model in some way. When the model is changed, an update function is called for the View (usually through an observer) and the result is interpreted and rendered by the browser. This cycle will repeat itself every time a user clicks on a controller action. The important thing about this architecture is that it encapsulates the different components and the View never has access to the Model – creating a scalable and secure system.

Another example of Rails favoring convention over configuration is shown in the controller. Rails prefers to use RESTful URIs which correspond to HTTP's GET, POST, PUT, DELETE etc methods. In Rails, these RESTful URIs are implemented via CRUD – Create, Read, Operate Destroy. Bus Tracker supports these CRUD operations as well and utilizes them to create local instances of each bus and any data associated with it.

Some of the API features, specific to Bus Tracker, are also listed in Figure

1. The controller relies on fetching data from the WMATA servers which are accessed via a helper class - WmataHelper. WmataHelper fetches all the possible information and then the required information is stored locally using the Controller's CRUD operations. For example, a new bus can be made with the create() method and then assigned the data from the API call or an existing bus can have it's information updated with the update() method. The locality reduces latency and WMATA server usage. Another API that is being utilized in Bus Tracker is Twitter's Bootstrap 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.

Model Relationships

Bus Tracker will contain information regarding all the buses in the DC metro area which can be quite a lot at times. The final model will consist of many relationships that tie the buses to physical locations on the map. Figure 2 shows the expected final relationship scheme for Bus Tracker.

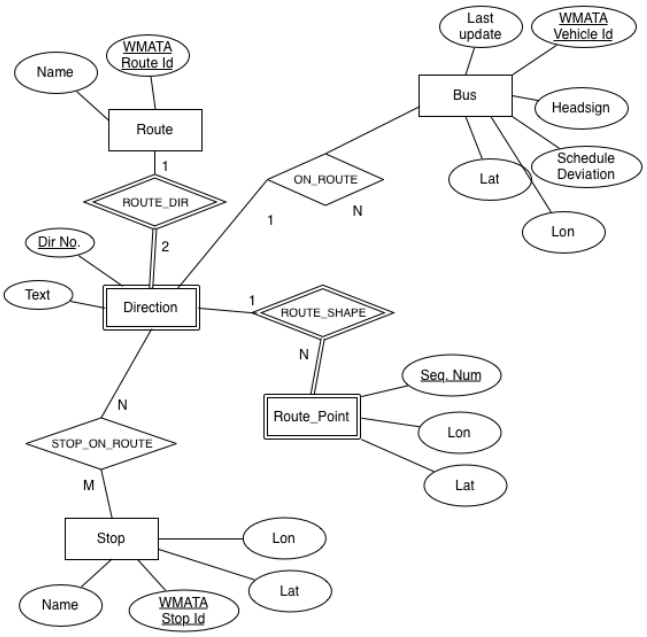

Figure 2: Model relationship diagram

This diagram shows the Bus model and all of the relationships necessary to provide a user with the best information available. As a brief overview, each bus has certain characteristics regardless whether it is en route or not. If the bus is en route, there is some more information needed such as what direction it is headed in, what route shape is displayed on the map, and what are the defining characteristics of the next stops. This is the information that will be stored locally after retrieving the information from WMATA.

API integration

To complete this project in a timely fashion, multiple APIs had to be utilized. As mentioned before one of these APIs is from WMATA which pulls the bus information. Another API that is used to display the map itself and the data on the map is Google Maps. Figure 3 shows an overview for how these APIs connect to the web server.

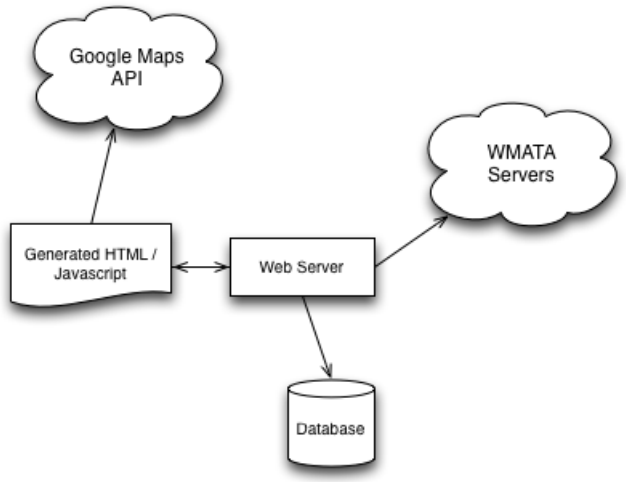

Figure 3: API integration overview

The web server begins this process by sending a request to the WMATA servers. Once the information is received, CRUD operations are used to store this information in a local PostgreSQL database. This information is sent to the View and lastly rendered using the Google Maps API. Google Maps provides a unique interface which makes it possible to display coordinates for

buses and their routes. Figure 4 shows how the information is pulled from the WMATA servers for display on the map.

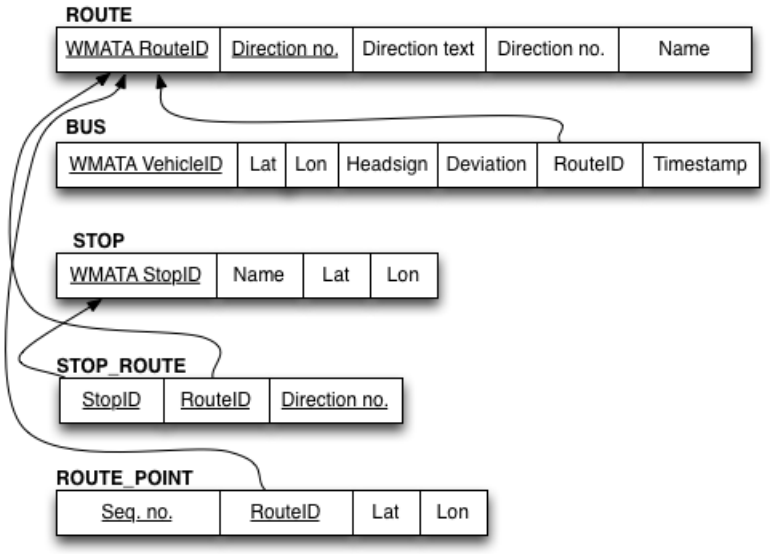

Figure 4: Schema for WMATA API

Using this information, all the bus information can be connected back to a certain bus using the route ID. The local database also mimics this structure to make the conversion easier.

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 geolocation 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]
5. Layer toggle widget that shows Google traffic overlay, bus markers, and stop markers [Will]

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. Add iOS/Android functionality using mobile web browser [Cluck]