Traffic Monitoring System

Technical Documentation

Group #7

Matt Araneta, Kevin Hsieh, Peter Lin, Geoff Oh, John Reed, Michael Simio

April 5, 2013

Website-based Application Descriptions

List of files and their basic descriptions:

File name	Short description
index.php	Home page, short introduction
header.php	Constructs head of web page
footer.php	Constructs end of web page
menu.php	Constructs menu bar
progress.php	Displays table of progress
repxxx.php	Displays corresponding report pdf
groupMember.php	Displays group member
map.php	Displays input to construct map
mapImaging.php	Server-side code of map.php constructs map
sqlQueries.php	Functions to connect to database
polyEncode.php	Not complete
css/main.css	CSS document for all webpages
css/menu.css	CSS document to style menu bar
Images folder	Contain images for menu bar
Pdf folder	Contains all report pdfs
scripts/jquery-1.9.0.min.js	Jquery source code
scripts/menu.js	Menu source code
scripts/pdfobject.js	PDF viewer source code

Full Descriptions

index.php

This is the homepage. This contains only a short intro.

header.php and *footer.php*

These files construct the beginning and end of each web page, to assure consistency between pages.

menu.php

This file contains the menu's html seen on the website.

progress.php

This file constructs a table of our current progress and archives of our reports in a table format.

repxxx.php

These files build the page to display our reports in a PDF viewer.

groupMember.php

This file displays the group members.

map.php

This is the page that displays the input for the user. The inputs are within a form, and on submit of the form it uses Jquery AJAX to perform a post back with the values entered by user. The inputs are a text box limited to numbers and two dropdown select menus for weather and time. The response of the post back is shown in the div formResult and shows the intensity color chart. This page contains a zoom in and out feature that is not fully debugged.

mapImaging.php

This is the server-side file that processes the post back from map.php. It will call a function defined in another file with the post back values. From the function, it will fill an array that contains the longitude and latitude of each point to construct and the point's severity. This file adds each point and color severity to the url to Google maps. With the url complete the file builds a html img tag with the src set to the url and returns the tag to map.php. It also returns a label for the not yet implemented zoom feature.

sqlQueries.php

This file will be for all functions that require a database connection. The function in the file called mapTrafficPoints. The function takes a weather, time slot, and zip code. It opens a connection to the database and performs a select query based on these conditions. It will access the Traffic_Incident table for the longitude, latitude, and traffic severity. It takes the results and parses it so that it is an array of arrays of longitude latitudes and severity color. It then closes the database connection and returns the result array.

<u>Mobile-based Application Description</u>

Basic Description

postData()	Communicate with web service

Full Descriptions

postData()

This function is called when the user presses the “Get Report” button on the get traffic report screen. It obtains a map image with traffic intensity overlay from the webservice and returns it on an Android ImageView on the lower half of the screen. The function first obtains the data from the fields entered by the user (zip code, time, and weather), and creates an Android Http Client to interact with the mapImaging.php part of the web service. The response is an HTML tag. The postData function gets the message of the body and converts it to an InputStream, which is a stream of bytes. This stream of bytes is read into a StringBuffer and converted into a String data type. At this point, the content of the String is the text that makes up the HTML tag. A URL pointing to the map image with overlay is parsed from the String and the image is returned to the user.

Data Collection Documentation

We currently have three functioning data collection scripts. Our Weather script gathers data automatically from the Wunderground API. First, it reads an input file of zip codes. Weather information is accessed from the API by changing the zip codes in the url through string manipulation - by looping through all the zip codes in the input file, weather information can be accessed for any zip code we desire. In order to obtain the information for each zip code, we extract JSON object information and store it into a hash table.

The NJ511 script gathers data from the RSS feeds on 511NJ.org when it is run. A list of updated traffic incidents are kept in an XML file.

The scripts for extracting data from the Bing Maps API and the Nokia Maps API are still works in progress.

Database Implementation

The APIs grant us access to various pieces of information. However, we have to ensure that the data we are attempting to store in the database is consistent with the database columns. For example, our database stores an integer called Date_Time, which is an integer from 0-7; 12-2:59AM = 1, 3-5:59AM = 2, ... , 9-11:59PM = 7. When we obtain the time observed from our Weather API, the get a string saying something like "Last updated on 11:29 PM". Therefore, we implemented several subscripts which convert the data we receive into data which our database requires. In our nj511 script, we obtain a 'coordinate', which is a string containing both latitude and longitude. Our database requires these values separately, so we have subscripts which convert the coordinate into separate latitude and longitude values.

After collecting the information we need, we have to store the information we need in our database. To do this, our Perl script must be able to connect and interact with our database. We do this by implementing the Perl Database Interface, or DBI module.

The DBI module allows us to connect to a database, using the appropriate Database Driver (DBD). Currently, we are using a local Microsoft SQL database, and thus we are using a driver called OBDC.

This allows us to create SQL queries in our Perl script and use the DBI to connect with our database and execute them. When we connect using the DBI, the Perl script sends the query to the DBI, which uses the appropriate DBD to translate the query into the format requested by the database. The database executes the query, and the DBI returns any result back to the Perl script.