LocoMotive: Product Guide

COS 333 Final Project – Spring 2016

Brian On, Trey Todnem, Adam Berman

May 10, 2016

1. User Guide

1.1 How to get our app

We developed LocoMotive as a hybrid mobile application within an Ionic Framework so that we would be able to have our program work on both Android and iPhones. However, publishing applications for Android requires less money and less time, so the application is currently only available on Android through Google Play, but it will eventually be put on the Apple App Store as well.

To get LocoMotive, search for "LocoMotive Transit Group" on the Google Play Store. LocoMotive is the application name, and Transit Group is the developer name. Install this from the Google Play Store and then you should be ready to start our application.

1.2 How to use our app

Home Screen:

We'll start with the Home screen: when you first start our application, you will be taken to the home screen. From here, you can select an origin, destination, and date for which you want results. Pressing the blue button at the bottom will bring you to Search Results. There is also an Options Button in the top-right corner. From the Options Button you can go to Notifications or Visualize.

Search Results:

After you press the blue button, you will be taken to the Search Results. If there is no internet available or if there is no data available for this date, the screen will say "no data available." Otherwise, this screen shows a "card" for each possible travel route throughout the day, sorted by departure times. These cards display only the essential bits of information for a given route: the departure and arrival times, the number of transfers, and overall duration of the trip. Selecting one of these cards brings up two new buttons above the given card: Live Tracking (left) and Information (right). Selecting information will bring up a window that shows the details of the selected route, including each transfer and its corresponding arrival time, departure time, and train number. Selecting the Live Tracking button will bring up a new screen Live Tracking details.

Live Tracking:

Live tracking also requires an internet connection. It depends on the New Jersey Transit website for information, so there may not always be live tracking data available. In general, the site up is except for a couple hours at night (1 AMish), when users should not be using the feature anyway. If there data, however, it will be shown here. When this screen first opens, you will see all the train stops along the current route. If there are one or more transfers, you will be able to select among the different trains along that route using the Live Track Bar on the top of the screen.

Below each stop is the time of arrival/departure or status of the train. If a valid time is given in the live tracking data, you will be able to set a notification for that stop by using the plus sign on the right side of the screen. You have the option to set a notification for anywhere between zero and twenty

minutes before arrival at the given station stop. This notification time will be automatically updated as live tracking data changes, so delays will not be a problem for the notification system. Other possible statuses for trains include: ALL ABOARD, BOARDING, DEPARTED, ON TIME, etc.

Notifications:

You can see your current notifications at any time by selecting Notifications in the Options Bar in the top-right corner of any page. All notifications that have not already expired will be located on this screen. You may have a maximum of five notifications at any given time. After selecting a notification, you will see two buttons: Live Tracking and Delete. By selecting Live Tracking, you will be taken to the corresponding Live Tracking data for which the notification was made. By selecting Delete, you will be given the option to cancel your notifications. You may also cancel all notifications by selecting the Delete All button at the top of the Notifications screen.

Visualize:

Our final feature is visualize. By selecting Visualize in the Options Bar, you will bring up a map that shows the route you have selected. You will see you origin, your destination, and the route you will be taking between them. If your phone currently has location services enabled, you will see your current location by looking for the person icon. Your location is automatically updated as you move towards your destination.

2. Frontend Developer Guide

This document gives an overview of the systems used for the development and production of our application, LocoMotive. We will discuss libraries, technologies, and high-level methods used to develop our application in regards to the Front-end, Back-end, and Scraping.

2.1 Front-end

The front-end of our application consists of our hybrid mobile application, which we developed within an Ionic Framework using classic web technologies HTML 5, CSS, Javascript, and the Javascript framework AngularJS. Ionic leverages Cordova, which is an open-source mobile development framework to package up the HTML, CSS, and Javascript in native skins for deployment to Android/iOS platforms.

2.2 Views and Routing

We developed our application within the Ionic Framework, which helped to layout the format of our frontend directory for us. We mainly developed our application in the *www/* subdirectory which contains the HTML templates for all our pages/views (as well as our JavaScript code). The HTML file that sits by itself in the *www/* directory, *index.html*, is the first page that loads when the application starts up. Given our goal to create a clean and straightforward application, we knew the routing layout of our website would not be overly complex and could be single page application. Consequently, we used the simple approach of using *index.html* to contain the parent view (the *templates/background.html* HTML template) for our application, with each individual "page" (template) in the application as a child view displayed through the parent view (nested views). Using this approach, we used the parent view to store HTML elements that would be consistent and uniform throughout the pages of the application; each individual view could then customize its respective page further per its specific purpose. Accordingly, the parent view template *templates/background.html* in *index.html* simply stores the template for the options menu (from which you can navigate the *Home, Notifications, or Visualize*) that opens up when the Options Button is tapped.

We then created HTML templates for each individual "page" the user would see that could be served up when the appropriate button was clicked. These are stored in the *www/templates* and consist of templates for the *Search Results* page, *Live Tracking* page, *Notifications* page, etc. We note that we also create separate HTML files for individual popups that require a significant amount of HTML, e.g. the Information window that pops up from the *Search Results* page.

We used Angular UI-Router for routing instead of the normal *ngRoute* method. In general, this approach allows one to change application views based on the state of the application and not just the route URL. We used UI-Router's *.config()* function to define routing relationships, making *templates/background.html* as the home state *app*. Then, using dot notation, we established parent-child relationships between home state and each child state (each child state is the home state with a different child view i.e. showing a different HTML template/page). Accordingly, the child state corresponding to the actual home page (i.e. the search page) was named *app.home*, the child state corresponding to the *Search Results* page was named *app.schedResults*, etc. The config function that defines the routing is stored in the JavaScript file *app.js* in the *www/js* directory.

2.3 JavaScript and AngularJS

We implement the business logic of our application in JavaScript with an AngularJS Framework. The main feature of AngularJS is the use of controllers, which allows us to generate behavior behind DOM elements. Given the single page design of our views, we opted for a simplistic, cross-template single controller design for our application, with various functions split into services that are "injected" into the controller for code-readability. In general, AngularJS services are substitutable objects that are "wired together" into controllers/other services through "dependency injections" (inclusion of the object at instantiation of the controllers/ other services) and allow us to modularize our code.

The code for our main controller is contained in *www/js/app.js* and the controller is named *starterCtrl*. The *www/js/* directory also contains the files containing the JavaScript services for each of the controllers we use. We have four services:

- *data_service.js*
    - contains functions for HTTP requests to the backend for train schedule data and live tracking data that return promises.
- *map_service.js*
    - contains all functions related to our visualization (map) feature, including functions to initialize the map, display the route between an origin and destination, GPS tracking, etc.
- *notification_service.js*
    - contains functions related to setting notifications and updating currently set notifications.
- *time_service.js*
    - contains functions for date/time manipulation as native JavaScript objects or strings. Given the nature of our application, several time functions were necessary to validate time strings, extract data from date objects, etc. Some of these functions were decidely more specific: for example, the live scrape data does not include AM/PM in the arrival times for a given train (which are non-military). Consequently, we include a function in this service that uses the time from the train's corresponding non-live schedule to assign AM/PM to each arrival time

.

Our main controller uses these functions to create the overall functionality of the application. For example, searching for schedules on the home page prompts to controller to call function in our *data_service* service that makes an HTTP request to our backend database for the necessary data.

Our controller makes extensive use of the built in AngularJS directives like *ngClick, ngBind, ngRepeat, ngInit, ngModel*. Combined with the AngularJS ability to directly display JavaScript variables in HTML use double brackets, these features allow for automatic data synchronization between our controller and views. For example, *ngClick* allows us to trigger the function that requests schedule dat when the user taps the Search Button on the home page. After we obtain requested schedule data from the backend, we can directly display the data using *ngRepeat* and double brackets in the *www/templates/schedResults.html* template to dynamically generate HTML elements (the "cards") through iterating over each schedule object without boilerplate JavaScript.

2.4 Cordova Plugins

We use a couple Cordova plugins to enable our hybrid mobile application, built in web

technologies, to access the native features of the phone:

- Cordova Local Notifications
  - *https://github.com/katzer/cordova-plugin-local-notifications*
  - We use Cordova Local Notifications to schedule, view, cancel, and update local notifications on the phone for a given stop. The code using this plugin is contained in the service *notification_service* in the *www/js* directory as well as the controller code at *www/js/app.js*.
- Cordova Background
  - *https://github.com/katzer/cordova-plugin-background-mode*
  - We use Cordova Background in order to consistently update notifications set by a user to ensure they are triggered at the appropriate times. For example, if there is a delay after the user sets a notification, we would be able to catch it by continuously checking the live data page on the NJ Transit website and adjusting the trigger time of the notification. Clearly, our application would not be able to update notifications if the user exits the application (the notification would still trigger though). However, in the case that the application is not exited but simply running in the background, we would like to still be able update notifications. Because Android/iOS platforms pause apps in the background, this would be difficult to do. Consequently, we use Cordova Background to enable our application to "unpause" while in the background to constantly scrape live data (every ~2 min) and update appropriate notification times. The code using this plugin is contained in our main controller at *www/js/app.js*.
- Cordova Geolocation plugin
  - *https://github.com/apache/cordova-plugin-geolocation*
  - We used Cordova's Geolocation plugin to access the native GPS features of the user's phone. This was essential for our visualization feature to properly work (track the progress of the user).

2.5 Google Maps and Google Places

For our visualization feature, we used an assortment of Google services already available to JavaScript development. The code for our visualization feature is contained in the service *map_service* in *www/js*.

- *Google Maps*
  - We used the Google Maps Javascript API to get the actual map on which we would be showing routes and positioning. We had to register with Google to get an API key. According to Google, the key can be used for free until exceeding 25,000 map loads per day for 90 consecutive days.
- *Google Places*
  - Here we used the Javascript API for Google Places. We had to hard-code the possible locations into the file to make sure the correct locations were used for directions. We have Google Places API draw the route on the Google Maps map, after correcting the names given from the front-end to the names that will be correct on the map.
  - This feature is free for up to 2,500 requests per day, and $0.50 for each 1000 requests over 2,500.
  - This feature could be hard-coded as an overlay, but this way was much more simply implemented.
- *GPS Tracking*

    - GPS Tracking is done using a constantly-updating method from Google Maps. We had the option to instead implement the tracking to update every minute or two, but we decided it was not too much of an issue to run the user's battery down a little more.
    - We draw the user's location using a hotlinked icon from a free-for-commerical-use icon website. We thought it was safe to assume the user would have internet if they have location services working.
- *Custom Map Coloring*
    - We implemented a custom map-coloring for our Google Maps map that we felt fit with the style and colors of our application better that the vanilla map that loads from Google Maps. We found the basic coloring on a site that offers free-for-commercial-use colorings for Google Maps.

2.6 Local Development and Deployment

Ionic comes with several features that allowed us to easily test our application. It also leverages Cordova to wrap our application in native skins for deployment to app stores.

- *$ ionic serve*
    - allows us to test our application locally in a web browser.
- *$ ionic run android -l -c*
    - allows us to deploy our application to a phone connected to the computer for testing of certain features that do not work within a browser (e.g. local notifications). the *-l* flag stands for *live reload*, which means the application version on the phone automatically updates each time the code is saved on the computer, saving us the trouble of relaunching the application after each alteration. the *-c* flag prints console log statements to the terminal of the browser, which greatly facilitate debugging.
- *$ cordova build --release android*
    - allows us to generate a release build of our application for Android.

3. Backend Developer Guide

We stored all of the future route information in the back-end. We also pushed the load of live scraping for the client devices onto the back-end because it was infeasible to do the live scraping on the client devices in Javascript.

3.1 Future Scraping

This manner of scraping is used to populate our database, collecting schedule data starting from the following page:

http://www.njtransit.com/sf/sf_servlet.srv?hdnPageAction=TrainTo

If you go to this link, you will notice that the resulting page is actually a routing page: selects an origin and a destination from drop-down selectors, picks a data from a date picker, and then clicks submit to retrieve the desired train schedules from that origin and that destination. What the future scraping component of our back-end (primarily scrape_functions.py, update_database_adam.py, and super_scrape.py) does is use a Selenium webdriver, made headless via PhantomJS, to input an origin, a destination, a date, and hit sumbit for us to get to the proper scheduling page to scrape. The reason we

do this is because the scheduling pages all use the same generic URL, so we have to web drive to access the HTLM containing the data that we want.

Our web driving and the bulk of our scraping is done in scrape_functions.py. Web driving in particular is performed by the getRouteHTML() function in this file. It is performs a pretty standard Selenium web driving operation, first selecting the desired origin (one of the arguments) from the drop-down bar, and then doing the same with the destination (another drop-down). To select the desired date from the date picker, the function first opens the date picker, then it uses a for loop to toggle to the correct month. Next, it uses a partial xpath ("//*[@id='dp-popup']/div[2]/a[2]") to narrow down the resulting calendar table's HTML to each of the possible days. The function then uses a for loop to iterate through each of the days, selecting the desired one specified by the date argument. Finally, the function hits the sumbit button and then collects all of the HTML of the resulting page, returning this.

After this is done, our actual scraping process is ready to commence. The function getTrainData() is used to call getRouteHTML() to first collect the HTML of the desired scheduling page. Next, it creates a BeautifulSoup object with this HTML. The function first uses BeautifulSoup to collect all of the HTML withe the "span" tag, as this will isolate the actual table object containing all of the data. Next the function looks specifically into this newly isolated table HTML and runs past the first row of the table (the one containing the headings "Origin," "Departure," etc) since this is not the actual data we are trying to retrieve. Next, the function uses a large for loop (for item in g_data:) to run through the remaining HTML to find the actual data and store it in a large Python list, in which one element in this large list represents one column of the table (and therefore one "trip" from origin to destination, including all transfers, train numbers, and times). This for loop uses one iteration for each column of the table, using the order of the words in the table ("Arrive", "Depart," etc.) to determine which piece of information from each column belongs in each catagory of one "trip" element of the big Python list. Within each "trip" element in the big list is an inner list containing a collection of specifically organized internested lists and Python dictionaries (dicts) to organize the data in a JSON-friendly way. Here is an example of one "trip" element of the big list, all for a trip from Princeton to New York Penn Station:

```
[[u'04:58 AM', u'Princeton Shuttle #4106'],
 [[u'A', u'05:03 AM', u'Princeton Junction'],
  [u'D', u'05:20 AM', u'Northeast Corridor #3910']],
 u'06:12 AM',
 u'74 minutes']
```

Note that the u's before each string in the list simply indicate that they are unicode strings, which can be easily converted to any other string format one fancies. We see that it is a list of three lists and two strings, where each inner list or string represents one of the four main parts of the trip, containing all of the pieces of information which make up that main part. For example, the OriginDeparture main part (the first component of the trip list above), consists of a list containing '04:58 AM', the time of departure, and 'Princeton Shuttle #4106', the train of departure, which are the two pieces of information found in the OriginDeparture column on the NK Transit website representation. On the other hand, '06:14 AM', the penultimate element of the above trip list, is just a string, and represents the DestinationArrival column (for the first trip) from the NJ Transit website. Finally, it should be noted that the 'A' and 'D's that you see in the trip list represent "Arrive" and "Depart" transfers, respectively.

Once we have run through every column of the table, completing our big Python list with

elements of the structure shown above, the function finally makes a call to our getTripJSON() function to straightforwardly convert the list to a valid JSON object, which is returned.

To actually run all of this code for each pair or origin and destination we support, we use update_database_adam.py, which stores all valid origin and destination pairs in a Python dictionary (which keeps all origins as keys of strings, each with a corresponding value (which is a list) containing all supported destinations from that particular origin. This data organization is important because not every possible pair of origin and destination have a valid train pathway between them. update_database_adam.py uses a for loop to iterate through this dictionary, running updateRouteData() on every valid origin and destination pair within it, for today and four days into the future. Because of all the computation required for each iteration of this for loop (it involves web driving, scraping, and HTML processing), the program uses time.sleep(1) to sleep for a second between various parts of the for loop. This slows the process down, but prevents the program from crashing, which tends to happen without these resting seconds.

update_database_manual.py is used to manually add new data. We also have a file *super_scrape.py* which does what *update_database_manual.py* does automatically, maintaining 55 days into the future of data in the database at all times. It does this by updating one new day's worth of data 55 days ahead once a day. To do this, the program uses an infinite loop that uses time.sleep(86000) to run approximately once per day, catching any errors or crashes with try-catch statements to keep the loop running even if the web driving and scraping causes a crash (which sometimes happens). The data is actually put into our SQLite database using our updateRouteData() function, which puts the JSON into our Amazon server back-end.

3.2 Live Scraping

The Live Scraping feature gets the station stop data for a given from a page on the NJ Transit website that is updated in real-time:

http://dv.njtransit.com/mobile/train_stops.aspx?sid=PJ&train=<train_number>

We used BeautifulSoup in Python to get the data from the html that we got from the static website for the station stop data. The static website URL is created from the train number, which our application frontend specifies when the client device requests the data. The request goes through our Django application (discussed later), which performs the scraping after receiving the request using the following algorithm:

- We separated html paragraphs and then checked to see if " at " was in the paragraph.
- If it was, the string after " at " was the time at which the train would arrive, and everything before " at " was the station name.
- If it was not, we used the <i> section to find out the train's status and remove that from the entire paragraph's string to get the station name.
- If any of the data was not in a format we expected, we skipped that line and then all of this was returned to the front-end in JSON.

We then make this data available in a view on our application.

3.3 Django and Relational Databases

We used the Django web framework to create a simple web app that generates REST endpoints from which our frontend hybrid application can request/get the scraped train schedule data from our SQLite database. The data consists of two models (Python dictionaries that we can then serialize into

JSON objects for the frontend). The first model, *Trainroute*, corresponds to a schedule for a given route and contains various attributes such as the origin, destination, date, train name, departure/arrival times, etc. The second model, *Transfer*, corresponds to a transfer for a given schedule and contains attributes like arrival, departure times, etc. Because one *Transfer* can only belong to one *Trainroute*, but one *Trainroute* can have many *Transfers*, this is a classic many-to-one relationship in SQL databasing and we implemented it by creating two tables in our database -- one corresponding to each model. The many-to-one relationship is represented through a the foreign key attribute in *Transfer*, which stores the primary key of the *Trainroute* to which it corresponds. Incidentally, we generate unique primary keys for each *Trainroute* instance through concatenation of several fields that, when combined, uniquely distinguish it from other *Trainroute* instances: origin, destination, search date, and departure time.

We used the Django app Django REST Framework to facilitate the extraction/storage of data from the database and conversions between our two models as native Python objects and JSON objects for/from the REST endpoints. This was accomplished concisely through the creation of Serializer classes using Django REST Framework's ModelSerializer class for each of our models, which generates create/update functions that allow for seamless back-and-forth translation of data from our SQLite database to Python objects to JSON objects.

Finally, we created the actual REST endpoints as API-based Django views. We created functions in *trainsched/views.py* that handle various GET/POST/DELETE HTTP requests for our two *Trainroute/Transfer* tables using the Serializer classes we created for their respective models. For example, we wrote a function that handles GET requests for *Trainroutes* with a certain origin, destination, and search date (specified through request parameters). We then made these views available for interaction with our frontend application and scraping scripts (to post new data) by mapping each one to a URL using Django's regular expression URL scheme in the files *king/urls.py* and *trainsched/urls.py*.

3.5 Django Database Management and Local Development

Just like the Ionic framework we used on the frontend, the Django framework comes with several commands that enable easy development and edits.

- *$ python manage.py makemigrations*
    - creates new migrations based on the changes we made to models (e.g. created new ones or edited the fields of old ones)
- *$ python manage.py migrate*
    - applies migrations created by the *makemigrations* command to the relevant tables in the database
- *$ python manage.py runserver*
    - allows us to test our application locally in a web browser

3.6 SQLite Database

We used the SQLite database for storing the two models in our Django application. Although, it is often good practice to use a more robust and large external database for commercial processes, we found that SQLite was practical for our uses for the following reasons (aside from the fundamental fact that it is a relational database):

- The database was store locally, so using SQLite reduced the latency of retrieving information for the front-end. This was especially important for our app because our app was made as a

sleeker and faster alternative to the NJTransit website and mobile application.

- The database can handle at least hundreds of thousands and up to one-million requests per day. Considering that this app is only meant to be used by Princeton University affiliates, this seems to be plenty.

3.7 Amazon EC2 Server

We host our Django application on an Amazon EC2 server, running Ubuntu, for the computational power of our backend. To run our Django application on the server, we used the "nohup" command to make the process continue running after we logged off the server, and we used the "&" command at the end of the statement to make it run in the background. Amazon offers 750 hours of instance usage per month for free for one year, so we took advantage of this free server. We used pip to install all of the dependencies that we needed for the other services in the back-end. We used Security Groups to make sure that other users do not try to access our Rest endpoints. Port 8000 is the only open port.