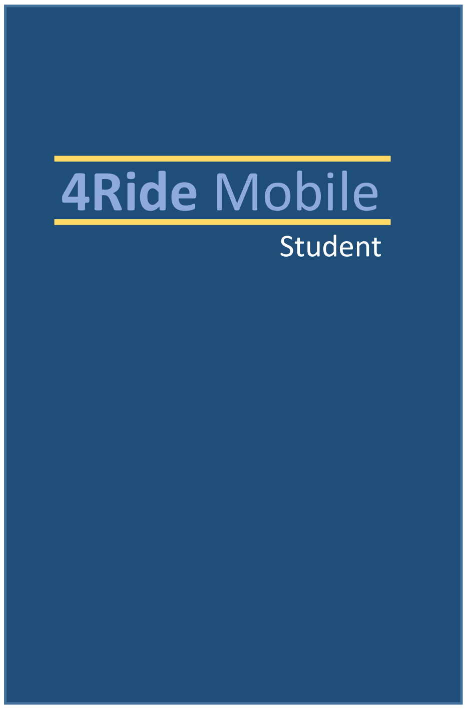
Randy Fitzmorris
Senior Design
Tuesday December 8, 2015

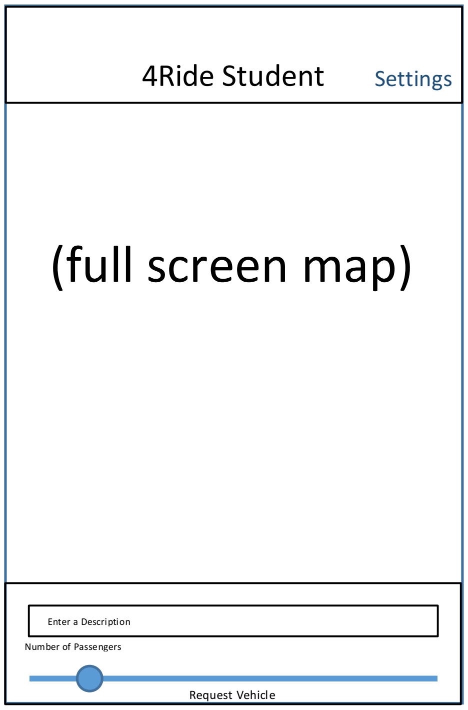
4Ride Mobile: Components and Interfaces

GUI Components – Client Application

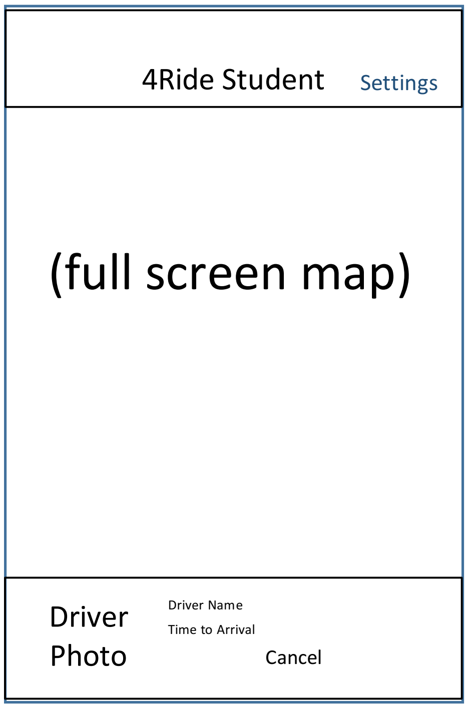
1



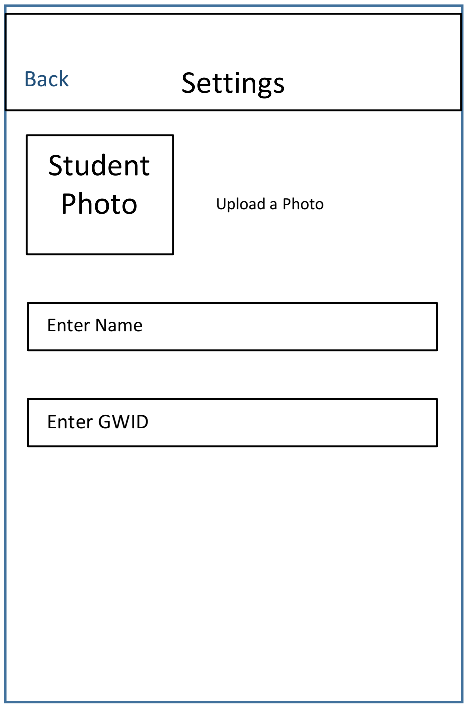
2



3



4



Client Workflow

The client app will immediately start with the launch screen (1) as the components are loading. The inactive map screen (2) will then be displayed.

When a user pressed “Request Vehicle” on screen (2), the view will shift to the active map view (3) to signify the request has been serviced. This view shows the vehicle assignment information. One the request if fulfilled (user arrives at location) or the user presses “Cancel”, the view will revert back to the inactive map view (2).

If the user wishes to enter or edit credentials, he/she can press “Settings” in either view (2) or (3) to access the settings view (4). The user can press “Back” at any time in this view to return to the previous view.

Client Components - Data Structures

Overview

1) Vehicle.swift Data Structure: A Swift class that will represent a vehicle pin plotted on the map view. This data structure will store the driver’s name as the pin’s title, capacity as the pin’s subtitle, and color (purple). It will also store the vehicle’s location in order to plot the corresponding pin. This data structure will adopt the Swift NSObject and MKAnnotation interfaces to conform to Apple’s MapKit annotations.

2) Credentials.swift Data Structure: A Swift class that will store the user’s entered data. This will include his/her GWID, name, and photo. This structure will be stored in the local file system when the app is closed used the Swift NSKeyedArchiver library and retaining a file descriptor.

Data Structures Pseudocode

Vehicle.swift

```
1) Initialize title, subtitle, pin color, and coordinates global variables

2) Initialize variables through constructor
```

Credentials.swift

```
1) Initialize GWID, name, and photo global variables

2) Initialize variables through constructor
```

Client Components – Algorithms

Overview

1) MapViewController.swift: This Swift class will contain the majority of the algorithmic functionality of the client application. This class will manage views (2) and (3) and corresponding functions, including current location, request/cancellation handling through the server web services, user input, and map management. It will implement the UIViewController interface to conform to an iOS view as well as the CLLocationManagerDelegate to manage the device's location stream.
2) SettingsViewController.swift: This Swift class will contain the algorithm handling user credentials. This class will manage view (4). It will implement the UIViewController interface to conform to an iOS view.
3) VehicleMapView.swift: This view will handle the management of adding and removing both pins and overlays on the full screen map from views (2) and (3). It will implement the Swift MKMapViewDelegate interface to accomplish this functionality.

Algorithm Pseudocode

MapViewController.swift

```
1) Initialize global variables for all UI components shown in views (2) and (3)
2) Initialize map
    a. Set default region of map
3) Initialize CoreLocation services
4) Query server for all vehicle locations using Alamofire (see external libraries below)
    a. Store vehicles as a Vehicle class in array
    b. Send array to Vehicle Map View to be plotted
5) Event Handler: request button pressed
    a. Get user defined destination, get origin from current location, get passenger
       count from UI slider
        i. Check if input is valid using geocoder
        ii. Check that either origin or destination address is within campus
            boundaries and other other is within 0.5 miles
    b. Convert origin and destination addresses to coordinates and sent to server via
       request web service call
```

c. When return JSON is received, remove current pins from map. Then plot a green pin for the origin location, a red pin for the destination location, and use the MapKit library to draw the route between them on the view's map.

d. Change view from (2) to (3).

e. From the confirmation JSON – plot a purple pin using the confirmation vehicle's location and update the driver's name and photo.

6) Event Handler: cancel button pressed

   a. Send cancellation request via web service to server

   b. When confirmation JSON received, switch from view (3) to view (2)

7) Event Handler: server pushes fulfillment update

   a. switch from view (3) to view (2)

8) Event Handler: settings button pressed

   a. Switch to view (4)

SettingsViewController.swift

1) Initialize global variables for all UI components shown in views (4)

2) Load user's current credentials from file descriptor, display on UI

3) Event Handler: "upload a photo" pressed

   a. Open camera roll viewer delegate from Apple external libraries

   b. Assign photo descriptor to global variable

4) Event Handler: GWID or Name textbox changed

   a. Update corresponding global variable

VehicleMapView.swift

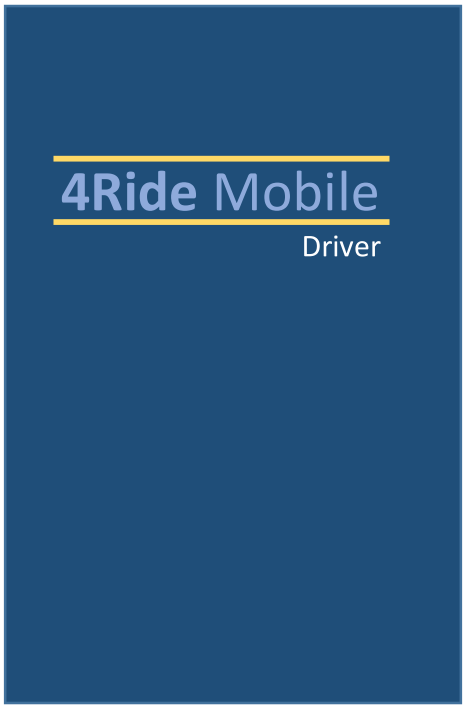
1) Add annotation function
   a. Assign corresponding values from Vehicle object
   b. Dequeue and reuse existing pin if available
   c. Add to map
2) Add overlay function
   a. Set color to blue
   b. Draw overlay from MKPolyLine parameter from MapViewController 6(c)

Client External Libraries

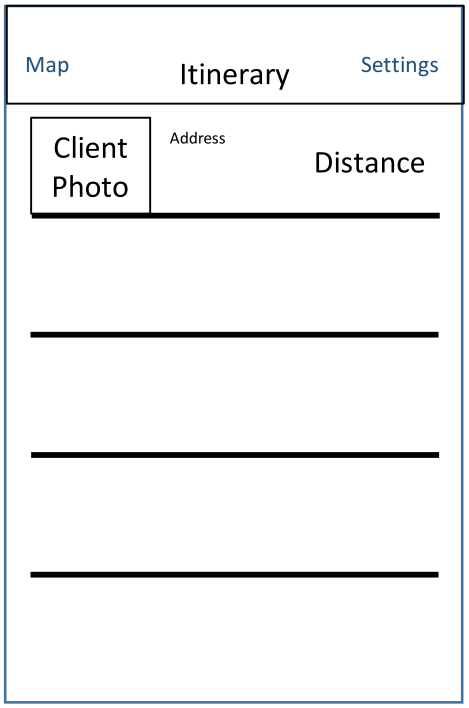
1) Alamofire: Swift library used to streamline HttpRequest protocol to server
2) SwiftyJSON: Swift library used to parse JSON responses from server

GUI Components – Driver Application

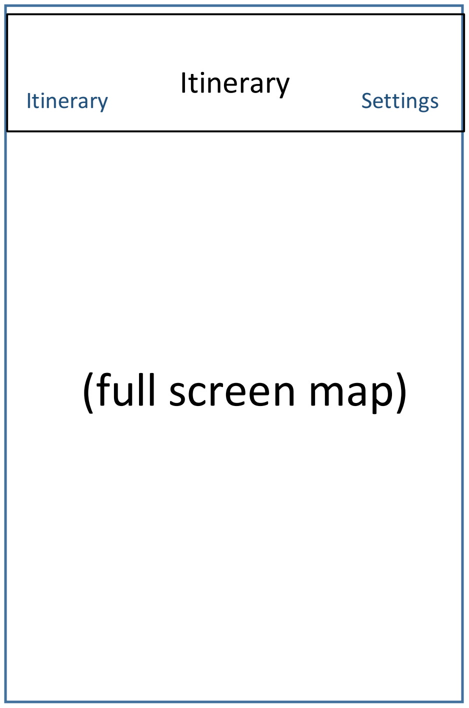
1



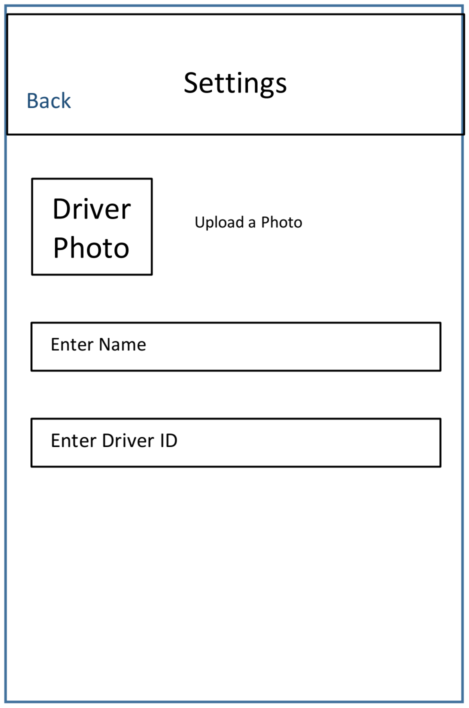
2



3



4



Driver Workflow

The driver app will immediately start with the launch screen (1) as the components are loading. The itinerary view (2) will then be displayed.

When a user pressed “Map” on screen (2), the view will shift to the itinerary map view (3). This view shows the vehicle’s current itinerary as a series of pins and a highlighted route. The driver can then press “Itinerary” to switch back to the itinerary map view (2).

If the user wishes to enter or edit credentials, he/she can press “Settings” in either view (2) or (3) to access the settings view (4). The user can press “Back” at any time in this view to return to the previous view.

Driver Components - Data Structures

Overview

1) ItineraryItem.swift: A Swift class that stores a photo, address, and coordinates of a pickup or drop-off point on the vehicle’s itinerary.

2) Credentials.swift Data Structure: A Swift class that will store the user’s entered data. This will include his/her driver ID, name, and photo. This structure will be stored in the local file system when the app is closed used the Swift NSKeyedArchiver library and retaining a file descriptor.

3) TableViewCell.swift: A Swift class that stores the data elements from ItineraryItem in a class the conforms to the UITableCell interface.

Data Structures Pseudocode

ItineraryItem.swift

```
1) Initialize Image, address, and coordinates global variables

2) Initialize variables through constructor
```

Credentials.swift

```
1) Initialize driver ID, name, and photo global variables

2) Initialize variables through constructor
```

TableViewCell.swift

```
1) Initialize photo, name, and location label global variables
2) Initialize variables through constructor
```

Driver Components – Algorithms

Overview

1) TableViewController.swift: This Swift class will contain the majority of the algorithmic functionality of the client application. This class will manage view (2) and corresponding functions, including current location and itinerary update handling through the server web services. It will implement the UIViewController interface to conform to an iOS view as well as the CLLocationManagerDelegate to manage the device's location stream.
2) SettingsViewController.swift: This Swift class will contain the algorithm handling user credentials. This class will manage view (4). It will implement the UIViewController interface to conform to an iOS view.
3) MapViewController.swift: This class will manage view (3) and corresponding functions, including current location and map management. It will implement the UIViewController interface to conform to an iOS view as well as the CLLocationManagerDelegate to manage the device's location stream.
4) DriverMapView.swift: This view will handle the management of adding and removing both pins and overlays on the full screen map for view (3). It will implement the Swift MKMapViewDelegate interface to accomplish this functionality.

Algorithms Pseudocode

TableViewController.swift

```
1) Initialize global variables for all UI components shown in views (2) and (3)
2) Initialize array to store itinerary items (ItineraryItem class)
3) Initialize CoreLocation services
4) Send "In Service" driver request to server, lock until response received
5) Initialize timer to poll for itinerary updates
6) Event Handler: timer tick
    a. Issue driver web service call, passing vehicle identifier as parameter
    b. When return JSON is received, empty the table
```

i. Check if any items match current location, if so then remove from itinerary and send verification to server

c. Reverse geocode all coordinates in JSON itinerary and package information as ItineraryItem

d. Reload table data

i. Repackage ItineraryItem as TableViewCell

1. Calculate distance from ItineraryItem location using currentLocation global variable

ii. Add cell to table

7) Event Handler: override pressed

a. Send override request via web service to server

b. When confirmation JSON received, remove item from array and call table reload function

8) Event Handler: map button pressed

a. switch from view (3)

9) Event Handler: settings button pressed

a. Switch to view (4)

10) Event Handler: app closed

a. Send "Out of Service" driver request to server, lock until response received

MapViewController.swift

1) Initialize global variables for all UI components shown in view (3)

2) Initialize map

a. Set default region of map

3) Initialize CoreLocation services

4) Initialize timer to poll for itinerary updates

5) Event Handler: timer tick

a. Issue driver web service call, passing vehicle identifier as parameter

b. When return JSON is received, empty the map

i. Check if any items match current location, if so then remove from itinerary and send verification to server

    c. *Then plot a green pin for the origin location, a red pin for the destination location, purple pins for intermediate locations, and use the MapKit library to draw the route between them on the view's map.*

6) *Event Handler: user presses "Itinerary" button*
    a. *switch from view (3) to view (2)*
7) *Event Handler: settings button pressed*
    a. *Switch to view (4)*

SettingsViewController.swift

5) *Initialize global variables for all UI components shown in views (4)*
6) *Load user's current credentials from file descriptor, display on UI*
7) *Event Handler: "upload a photo" pressed*
    a. *Open camera roll viewer delegate from Apple external libraries*
    b. *Assign photo descriptor to global variable*
8) *Event Handler: GWID or Name textbox changed*
    a. *Update corresponding global variable*

VehicleMapView.swift

3) *Add annotation function*
    a. *Assign corresponding values from Vehicle object*
    b. *Dequeue and reuse existing pin if available*
    c. *Add to map*
4) *Add overlay function*
    a. *Set color to blue*
    b. *Draw overlay from MKPolyLine parameter from MapViewController 6(c)*

Driver External Libraries

1) Alamofire: Swift library used to streamline HttpRequest protocol to server
2) SwiftyJSON: Swift library used to parse JSON responses from server

Server – User Interactivity (Web Service)

The client and driver apps will both interact with the server through the HttpRequest protocol, passing POST parameters to signify the nature of the request as well as any data associated with it. Each request will also include the app's user credentials, a CCA authentication key (see

Network Communication – Security section), the type of app connecting (driver or client), and type of request (i.e. pickup request or itinerary request). Any associated data will be send in the form of a serialized JSON with “data” as the parameter key.

Server – Demo

Since the server does not require a UI, it will be difficult to demo the underlying dispatching algorithm. Therefore, a web-based dispatch monitoring UI will be built in JavaScript to show active routes of all outstanding 4Ride vehicles on a map. This will include all pickups, drop-offs, pickup requests, cancellation requests, and driver overrides in real time. The JavaScript UI will utilize the Leaflet.JS mapping library to plot this information.

Server – Data Structures

1) Vehicle.java: This Java class will contain vehicle-related information on the server – total capacity, current capacity, driver name, location, node request order, a complete weighted directed graph to represent the relative difficulty to getting between two nodes, and the current optimal itinerary.

Vehicle.java Pseudocode

```
1) Initialize all components described above as global variables
2) Get all global variables from constructor except itinerary and request order
    a. Check that weighted digraph is complete
    b. Check that coordinates are valid through Google API geocoder
3) Get and set methods for all private global variables
```

Server – Algorithms

1) Servlet.java: Implementing getPost from HttpServlet interface, this java class manages all driver and client app requests. The servlet maintains an array of vehicles of type Vehicle. For clients, the server will fulfill “get all vehicles”, “pickup”, and “cancel” requests. For drivers, the server will fulfill “in service”, “out of service”, “override”, “and itinerary” requests.

Servlet.java Pseudocode

```
1) Initialization: Create vehicle ArrayList of type Vehicle that will can be dynamically resized
   for service changes.

2) Under doPost, parse incoming responses to determine request type, and call
   corresponding request handler (passing serialized JSON of associated data)

3) For each request, deserialize data JSON using Jackson library

4) Request Handler: Client Pickup Request
```

a. Reverse geocode destination and origin data using Google API

b. For all vehicles in service, add origin and destination nodes to the current weighted digraph

  i. Only do this if vehicle has capacity for the number of people specified in request

  ii. To determine weights between new nodes and other nodes, use a weighted sum of various factors

    1. Order requested (priorities stored as map in vehicle data structure)

    2. Time given traffic (distance matrix provided by Google Maps API)

    3. Likelihood of cancellation (post parameter, mapped by location, stored local)

    4. Likelihood of driver override (post parameter, mapped by location, stored local)

    5. Weather, time surges from Weather Channel API and traffic analytics

c. Compute optimal tour on all "new" graphs to determine which has the least optimal tour change

d. Assign node to graph with least optimal tour change

e. Recompute optimal tour and add to vehicle's itinerary

f. Return vehicle assignment as response to pickup request

  i. Serialize vehicle's object and send as JSON

5) Request Handler: Client Cancellation Request

  a. Find vehicle in array with client node

  b. For each node in weighted digraph, remove edge to client's node

  c. Recompute optimal tour and add to vehicle's itinerary

6) Request Handler: Client Get All Vehicles Request

  a. Serialize vehicle array using Jackson and send as JSON

7) Request Handler: Driver In Service Request

  a. Create new vehicle object

  b. Add metadata sent in request data JSON

  c. Add to Vehicles ArrayList

8) Request Handler: Driver Out of Service Request

  a. Find vehicle in array that matches

    b. *Remove vehicle*
    c. *For all nodes in vehicle's itinerary, run new Client Pickup Request*

9) *Request Handler: Driver Itinerary Request*
    a. *Find vehicle in array that matches*
    b. *Serialize vehicle's itinerary using Jackson and send as JSON*

10) *Event Handler: fulfillment event*
    a. *Find vehicle in array that matches*
    b. *For all nodes in weighted digraph, remove edge to fulfilled node*
    c. *Recalculate optimal tour and update itinerary from result*
    d. *Push fulfillment to client on next request*

Server – External Libraries

1) Jackson – A Java library that can serialize Java classes. Will be used to send update data from the Vehicles ArrayList to both the driver and client apps.

Server – External APIs

1) Google Maps API: A web service API that represents a number of Google Maps services as Java objects. This project will incorporate geocoding and reverse geocoding to convert coordinates sent in an app request to an address key, then back to coordinates on pushed updates. all communication is done in coordinates to prevent geocoding inconsistencies between Apple's MapKit and Google Maps. The API's distance matrix will also be used, returning a matrix of special and temporal distances between a given array of origin and destination nodes, for the distance weighting factor.
2) Weather Channel API: A web service API that also provides services as Java objects. This will allow current weather patterns to also be a weighting factor in the scheduling algorithm.
3) Additional external APIs on the server will likely be incorporated to provide weighting factors to the scheduling algorithm. However, these factors are being determined in coordination with systems engineers for another senior design project and thus cannot be planned for immediately.

Network Communications – Security

4Ride Mobile will maintain security using app authentication and user authentication. For app authentication, the server (running on Apache Tomcat) will be configured with Client Certificate Authentication and both iOS apps will adhere to this protocol through the NSURLConnection class in Swift. For user authentication, the server will verify a user's GWID or Driver ID before granting them respective access.

Network Communications – Communication Failures

1) No network connection: The user will be alerted with a UIAlert. If currently in client view (2), it will be up to the user to issue the request at a later time. If currently in client view (3), the app will automatically try to reconnect every 15 seconds until the a network connection is made. If the driver or client have outstanding requests and 5 minutes has elapsed without a response, server will cancel the request / remove the driver from service.
2) Server Crash: iOS apps both have asynchronous connection handler to prevent freezes. Apps will break from polling if server fails, alerting the user through a UIAlert. If client is in view (3) at the time of the failure, his or her request will be revoked client-side and will return to view (2).