UBER Web Application

-Shivam Bharuka

Uber is a web application which allows consumers with web access to submit a trip request which is then sent to Uber drivers.
The Uber web application is built on ruby on rails with a MongoDB database.

Advantages of using Ruby on Rails:

1) The Model View Controller architecture divides the work into three subystems.

Advantages of using MongoDB:

1) The uber model requires future proofing for scaling.
2) Allows fast updates on data because of real time data input/output from different users, places, drivers and pools.

Algorithms:

a. **Driver dispatch Algorithm**
   i. Get a list of drivers in an arbitrary mile radius. Use Djikstra's to compute shortest travel time from driver to user.
      1. Calculate edge costs by multiplying traffic delay time and mile distance of nodes (which could be popular intersections, landmarks, etc)
   ii. Select closest few drivers and send available fare request to them
   iii. Users may change the pickup point

b. **Surge Pricing Algorithm**
   1. Divide cities into surge areas with independent surge prices
   2. Database will constantly track most popular areas by calculation ride frequency of every area over time.
   3. Apply correlation between supply and demand, wait times, traffic, etc. to surge pricing
      1. Factors increasing surge price: demand, location type (commercial or downtown areas should have high prices)
      2. Factors decreasing surge price: supply, low ride frequency, suburb/residential area

c. **Split Fare Algorithm**
   1. Calculate and return the user's share of the fare if the users were pricked and dropped off at the same location

d. **UberPool Algorithm**
   1. Multiple travelling salesman problem
   2. Split the cost
      1. Each user pays for their share of travel time and distance.
      2. Calculate for each user their share of fare. Divide the distance traveled while sharing with any other user by the

number of users that distance was shared with, and add to distance traveled alone

MVC architecture:

Model: It maintains relationship with data and objects. Few models for authentication are:

1) User model

```
class User < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, format: { with: VALID_EMAIL_REGEX },
uniqueness: { case_sensitive: false }
  has_secure_password
  validates :password, length: { minimum: 6 }
end
```

2) Admin Model

```
class Admin < ActiveRecord::Base
  devise :database_authenticatable,
         :recoverable, :rememberable, :trackable, :validatable

attr_accessible :email, :password, :password_confirmation, :remember_me
end
```

3) Driver Model

```
class Driver < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, format: { with: VALID_EMAIL_REGEX },
uniqueness: { case_sensitive: false }
  has_secure_password
  validates :password, length: { minimum: 6 }
end
```

*Payment gateway for credit card, debit card and paypal will be added separately.

View: Interface of the program

1) User Interface: “Call a driver” interface
   1. User initiates travel plan
      1. Suggest destination based on history or input from other app
      2. Show offers/promotions
      3. Show traffic information for selected destination
      4. Add other users to pick up/drop off (for pool)
   2. Map
      1. Exact Location using GPS
      2. Request server for nearest available drivers, get their locations and plot on map
      3. Estimated fare using GPS via miles
   3. When Driver accepts
      1. Driver’s profile
      2. Track the taxi coming to you.
      3. Track while riding
      4. Option to cancel the trip.
   4. When travel is over
      1. Rate the driver
      2. Send an email with a receipt

3. Show metrics and fare divisions (between pool users)

2) Driver Interface
    a. Option to be active or inactive
    b. Notify the driver when ride is requested, option to accept/cancel request
    c. Show customer details, location
    d. Show user phone number and option to call from the app
    e. Calculate estimated travel time to user
    f. History with past journeys and revenue generated

3) Admin Interface
    a. Database of all driver's profile
    b. Database with all the rides taken and taking
    c. Database of every customer's profile
    d. Ability to message each driver and customer
    e. Revenue generated - daily, weekly and generated
    f. Each driver's rating and reviews
    g. Questions received
    h. Ability to remove drivers and customers- editing database

Controller: Requests API server exposes.

1) User
    a. addUser()
    b. updateUser(userfields)
    c. scheduleRide(gps_location, destination)
        1. searchNearestDriver(gps_location)- Get all drivers within the radius, calculate traffic approximation given location.
        2. estimatedFare(gps_location, destination)
        3. updateDriverProfile() – return driver location
        4. trackCab(gps_location, driver_location) – track the taxi coming to you using google MAPS API
        5. trackRide(gps_location, destination)
        6. reviewDriver()
    d. cancelRide()

2) Driver
    a. addDriver()
    b. updateDriver(driverfields)
    c. userPick(gps_location, destination)- find shortest path to the user
        i. UserInfo() – to call the user