BestFuel Architecture and Design Document

Dave Knight, d7knight: 20501210
Anthony Reed, adreed: 20326940
Yufei Su, y42su : 20379150
Kyle Platt kbplatt: 20423636

Architectural Description

BestFuel is a gas-economy finding and tracking application that allows users to find the most efficient location to fuel their vehicles at. It combines car data such as economy and fuel tank size with distances and pricing of fuel. BestFuel even takes this a step further and strives to be an all in one travelling app that helps you locate preferred stores and restaurants that fit with your gas finding needs.

A variety of Architectural Patterns were used to amalgamate into an efficient and proficient application for users, managers and developers.

Client-Server

The Client-Server architecture allowed us to partition workloads between several servers and the client's device. Using an active internet or network connection the client connects to a private server storing authentication data, profile data, and fuel station data, to google's servers to gather location, maps, distance, geocoding and places data as well as to another third party car database, CarQuery, to gather all vehicle information.

Layered

As a common architecture present in most mobile applications the Layered architecture is abundant in BestFuel. Although we do not follow a strictly-layered system our layers include the physical devices, data storage, storage linking, networking, session data, and user interfaces typically in that order. The physical device provides access to apparatuses such as camera and gps and passes data storage to linking and networking so the session layer can access necessary components. This is all finally passes through the session layer into the presentation or user interface layer.

Event Based

Event based Architecture is an obvious inclusion in any application with a user interface. All user input and interaction follows the event based architecture. There are also many external calls that asynchronously trigger events upon their return such as Carquery javascript calls, GPS connectivity as well as all Google calls.

Mobile Code

As discussed below in the NFP Testability, Mobile Code is used by incorporating portable code in the form of Javascript and HTML. Our app interfaces with multiple Mobile Code components on demand.

Publish-Subscribe

Publish-Subscribe architecture is used for much of the connectivity between code bases.The device gets notified of returns and called within Javascript and vice-versa. There are several network calls that notify components in the map, in user data and for the user interface.

Non-Functional Properties

Security

For the Non-Functional Property of security one way in which our app supports it is in our Authentication System. In our Authentication System we don't store the actual password in our local or remote databases instead we store the hash of user's password. Therefore users can be sure while using our app they have high security and their password is highly protected.

Another aspect of security that is often overlooked is that we are not storing any of the user location, search, or visitation data. It is often a concern for users whether they are being tracked or recorded and by only storing email and username we ensure that we do not have any of that volatile data.

Efficiency

For the Non-Functional Property of Efficiency there are many ways in which our app supports it. Our app uses a local database to cache user profile data from our remote database located on our server. Clearly caching the user data improves the efficiency of our algorithms since it is much faster to contact local database rather than the remote one. It allows for consecutive use in multiple sessions without the need for constantly logging in.

A second way in which our app supports the NFP of efficiency is by performing all time consuming API calls and algorithms on our server side of our app. Our remote database on our server stores station data that we scrape from msn autos. We are given the address of each station not the geo-coordinates and our algorithm requires these addresses to be converted in terms geo-coordinates. Our algorithm would be very inefficient if the client had to geocode each station address on the fly and calculate which stations were nearest to them, instead our server converts the addresses to geo-coordinates and caches the results in our database for future lookups. We even implement an SQL query that can efficiently select the closest 30 stations within a radius of your location. Instead of using google geocoder service client side we improved the efficiency of our algorithm by performing these API calls and algorithms server side.

Adaptability

Our app is intended for Android Phones. As you may know google play services is required for all android devices wanting to use any of Google's Android APIS. Devices such as the Blackberry Phone, Amazon Phones and possibly soon windows phones only support running android apps natively that don't use google play services. To support all of the android platforms we restricted our app to not use any google play services. As an alternative to using the Google Maps Android API for our maps page we decided to use the Google Maps Javascript API which is supported among all android devices.

Scalability

Our app is highly Scalable for the following reasons:

New Features: We have plenty of new features coming available like support for social networking and the ability to search for nearby food places, restaurants and hotels in addition to providing fuel related information.

Data Storage: We have plenty of data storage available on our remote server for new users. Currently we are only using 50mb of the 5gb total hard drive space available to our server.

Testability

Our app uses the Mobile Code Architecture by incorporating portable code in the form of Javascript and HTML for our maps page. By using the Mobile Code architecture we can test our map component on any device with a browser which clearly improves Testability.

Testability is also enhanced since our map component is located on the server side. Everytime we make a change to our map component we don't have to redeploy our android apk to our mobile testing devices. Instead our app automatically synchronizes with the latest revision of our map component located on the server.

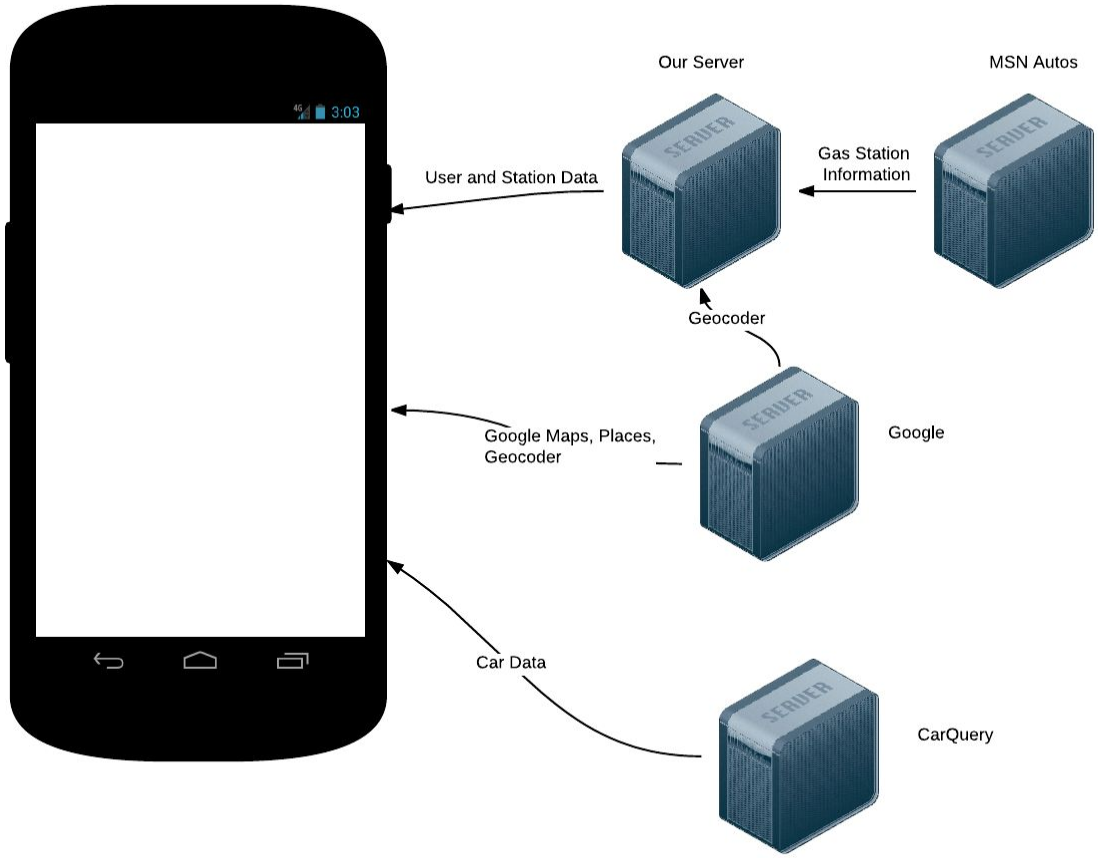
Dependability

Our app is highly Dependable since we gracefully handle errors such as invalid login or invalid profile options. If an error occurs while calculating your GPS location our app has the ability to manually recalculate your GPS location. We have also stress tested our servers to guarantee that increasing numbers of users and calls will not hinder the usage of our application.

Component Diagram

Component Diagram is complete we will merge the pdfs together later so that we get a wider diagram...

Physical Representation Diagram



System Design

Key Design Patterns and Data Structures

Observer Pattern

We use the observer pattern in many places throughout our application. One example is on our map page, to determine your location we make an asynchronous request to our MyLocation class. On a Location Update our MyLocation class notifies our Map class of the change through a callback function. So clearly the subject we are monitoring in this example is our MyLocation class and our observer is our Map Activity class.

Rationale

Clearly using the observer pattern here was a good choice since it provides low coupling between our MyLocation class and map activity. We wanted the best location data available, by using the observer pattern to wait for gps location data, our algorithms accuracy will increase which will be more useful to our users.

ViewHolder Design Pattern

Inside our profile page one design pattern we use is the ViewHolder pattern. The ViewHolder Design pattern is used to improve performance of ListView while scrolling it. Our profile page has a ListView of custom car views which we use the ViewHolder design pattern. In our Listview adapter class we have a ViewHolder for each car view which holds each of the views which we need access to make changes to the corresponding car view.

Rationale

ViewHolder is an obvious choice for displaying a dynamic listing of vehicles. We wanted the user to be in full control of the amount of vehicle data they could save and use which constituted using a ListView.ViewHolder, in Android, allows us to grab any listing of data directly without lookup. This aids in manipulation of data as well as the scrolling factor by allowing smooth efficient scrolling by not having to ever lookup or verify where each specific index of data is.

Model-View-Presenter Design Pattern

We use the MVP design pattern throughout our android application. One place where we use the MVP design pattern clearly is in our Profile page. Our views are the various views for displaying profile data on our profile page. Our model is our sqlite database class which stores all of the profile user data. Finally, our presenter is the Profile activity itself since it presents the data from our database (our model) inside our views. One advantage to using the MVP design pattern is low coupling between the model and views. Our Views can change without affecting our model and vice versa. This was very useful to our team since one of our team members could be working on the layout of our profile page view and another working on storing data inside our Database independent of each other which improved the performance of our team. This design pattern clearly promoted the NFP of testability since we could test our Model and View components separately.

Rationale

Model-View-Presenter as well as Model-View-Controller are abundantly found in the software world especially in user interface heavy applications. Model-View-Presenter was a better choice for BestFuel as having our activity level code interface into user interfaces, especially in terms of the map pages, allowed for safe interaction as well as a more loosely coupled reliance. This is also beneficial in our profile page car selection interaction. The view has control over what is happening and sends data to the application and then back. It is not fully dictated by a controller and has a one to one mapping to its components. Finally MVP is excellent for individually testing components without reliance on others and that works excellently in a team setting.

Facade Design Pattern

We used the facade design pattern to abstract various complexities throughout our application. Our app requires a mechanism to store and retrieve user data. On the Android Side of our app we use a local database to cache user data from our remote database located on our server. Here are the following tables we have in our databases:
User - stores user credentials
Preferences -stores user profile preferences
Car -Store a user's car information
Station - stores station information
Fuel - stores a fuel information relating to a station

We created a simple database access class with different methods to read and write to the different tables above. We used the facade pattern which hides the complex database access interface behind a few easy to understand and maintainable interface.

We also have a server-side database. To access this database we built a restful API. Clearly the facade design pattern is also being used in our restful API. Our restful API is a key component to our app and is used in almost every component. Our station algorithm uses our restful API to retrieve station data. Our login and registration components uses our restful API for authentication and user account creation. Finally our profile and map components uses our restful API for managing user preferences and car data.

Rationale

In our app, by using the facade pattern we have simplified the access to our complex database system and decoupled the rest of our app implementation from our database subsystem. Reusability of code is also benefited from using the facade pattern for our database interface since if multiple components require access to our database they can both use the same interface.

Composite Pattern

The profile page uses composite pattern. Inside the class for displaying each vehicle info - CarAdapter, class View serves as the Component class in a composite pattern. ViewGroup serves as the Composite class.The interface for each vehicle information object is declared in

ViewGroup class. The make, model, car, etc for each vehicle are placed in a ViewHolder. Each vehicle component consists of ImageView and Textview, which are the leaf objects to the component.

Rationale

Using composite class helps managing the vehicle information objects. Instead of manually position the object, each vehicle information can be added or removed in class View.

Station Algorithm for Selecting Cheapest Gas Station

The algorithm calculates the cost needed to get to each station and returns the station with lowest cost. Inputs to the algorithm are list of station prices, list of distance to each station, and fuel economy. The fuel needed is calculated by the percent of economy and distance to the station. The cost needed is then calculated based on the price at the station and fuel usage. The cost is then compared against each station to find the cheapest and most fuel efficient station to go.

Coupling and Cohesion Discussion

Our Design provides low coupling and high cohesion since of the various design patterns we employ. We discussed how the use of the Observer Pattern provides low coupling between our map activity (observer) and our MyLocation class (subject). We also discussed how using the facade design pattern provides low coupling between the database and our android activity code. Finally, we discussed how using the MVP design pattern provides low coupling between our models (database class) and views (android views).
Our application is highly cohesive since each activity does what it is supposed to do and does it fully. Our application is split into modules each which perform independent tasks to meet our apps needs. An example of high cohesion in our application is in our Login component. Our login component is self explanatory its job is to login a user into our system, there isn't any unrelated functionality in this class. Other functionality like registering for an account or managing your profile we have kept separate from our login component to support high cohesion.

Design Evolution

A proposed adaptation to our design includes the use of our algorithm for including restaurants and places to acquire food in addition to simply finding gasoline. While this increases the amount of information our app will need to store on our user, it does not overly complicate our design. Adaptations to our algorithm include increasing the range of items we include in our search criteria as well as allowing the user to specify and limit what is used.

We also envision our system being altered to support users updating gas price information. To meet these new requirements would be very easy. Since in our design we already have a database to store all our gas station data upon receiving an update request for a gas station price we could simply update the price in our database. We could also create a point system and reward users for updating gas prices similar to other teams.

Class Diagram

1. AppConfig
2. AppController
3. MyLocation
4. SessionManager
5. SQLiteHandler
6. Station Algorithm
7. StationData
8. LoginActivity
9. MainActivity
10. MapActivity
11. ProfileActivity
12. RegisterActivity

Scraper, Database Classes on the class diagram

4.Contributions to Design level Classes and Architecture

Kyle

- Front End:
    - Station Algorithm
    - Car Query API
    - Local Database
    - Profile Page
    - All User Interfaces
- Backend:

Dave

- Front End:
    - Map Component
    - Login and Registration
    - Profile Page: Car View
- Backend:
    - Rest API to database
    - Scraper Algorithm