Software Design Document

Team 7

Project Name: MapTak
Members: Travis Coria, Trevor Coria, Tylor Garrett, Michael Hockerman, Kyle Potts, Nick Stanish

Purpose: The purpose of this system is to provide event and building administrators with a convenient and fast tool for creating and sharing customized maps with their customers. A map consists of points of interest, called taks, with accompanying metadata.

Non-functional requirements:

1. System should be secure when handling data.
    a. The data users send to our service could contain private commercial information which shouldn't be shared with others. In order to obtain this aspect we must keep all data secure.
2. The Android app and website should be as functional and user friendly as possible.
    a. Users should be able to complete the tasks they want to do easily without in-depth tutorials.
3. Website should be platform independent.
    a. Any device with internet access should be able to use our service without any restrictions imposed by us because Administrators wish to target as many people as possible which implies they wish to target all available platforms.
4. API documentation for REST should be clear and readily available.
    a. Developers need a hassle-free way to understand how to access and extend our data.
5. Data on the server should never be manipulated without the administrator's consent.
    a. Administrators should feel secure with the data they put on our servers.
6. The server which hosts the service should be reliable and have little to no downtime.
    a. Users will be using our server to provide a service to their customers, and if our server is down this service can not be provided which may make customers unnecessarily upset.

Design Outline:

1. Client-server architecture (*with thin client)*
    a. Data is stored on the Internet and is quickly available to clients anywhere.
    b. Creating a client to connect to the server requires minimal work. For our

Android app there are many free HTTP Client libraries we are easily able to use.

c. The client-server model is popular, and Android Apps have inherent support for this kind of model and the problems that come along with it.
d. Thin clients can be quickly downloaded by users who are at events and are only interested in finding relevant information right at that moment.
e. Thin clients have minimal delays compared to fat client which will create many loading screens for the user. We want the user to be as free of loading screens as possible. Therefore a thin client seems like the best option.
f. The client-server architecture is a simple architecture that allows a computer with more processing power(the server) to be able receive and transmit data to a computer with less processing power(client). A majority of the computing will be done on the server end, therefore a thin client is needed. Our server will be receiving data from the Android app and storing the data. The server will also respond to the client and give it information that it requested.

2. REST API
    a. Uses HTTP so it can scale to many different platforms very well.
    b. Simple communication protocol, client needs only standard URI.
    c. Easy to document.
    d. The REST API is the best design that allows a client using the client-server architecture to easily communicate with the server. The REST API uses fixed URLs which are known to client to send requests to the server. Once a request is received, the REST API will send a response to the client in a specified format(HTML, JSON, etc). The purpose of the REST API is to make receiving and sending data to and from the server as easy as possible.
3. JSON For Communication
    a. JSON(Javascript Object Notation) is a specified way of serializing object in Javascript. The serialization is a text based dictionary format which is simple to read and manipulate. Our server and client will be sending data in the JSON format. The purpose of using JSON is to allow a simple way to be able to serialize all the data we are sending and receiving to and from the server.
4. Python Flask Web Framework
    a. The Flask web framework is a Python library that allows one to quickly create a web service. The purpose of Flask is to function as our web server which will handle all the requests that the server receives. Flask will also allow us to quickly create the URL's we need for our REST API.
5. Google App Engine
    a. We will be using Google App Engine to host our web server. The purpose of

Google App Engine is to allow our web server to be hosted on the Internet and easily accessible from anywhere. Google App Engine will also function as our web management software and our data storage (using its NDB API).

Interactions:

1. Client and Server
    a. The client will send a GET request to a URL. The URL will be specified in our REST API, and the server will be an expecting a GET request from this URL. Since it is a GET request the server will respond with the data (in JSON) and send it back to the client. Or the client will send a POST request to a URL with specific data (in url parameters or JSON). The server will process the data and send the response to the users.
    b. The server will be based on the Flask web framework and hosted on Google App Engine. Google App Engine will interact with our Flask framework and start and stop it as we please. Our server will also interact with clients by providing map data and updating the data stored on the server.
2. Flask and Google App Engine
    a. App Engine has a persistent data store system called NDB and an API in Python to access this NDB. Using the NDB system, with Flask we will be storing data in the NDB. The NDB is storing our app classes and their associative data.

Design Issues:

1. Choosing web framework and subsequent language
    a. There are hundreds of web frameworks and each of them has their advantages and disadvantages. Since we will be using Google App Engine, we are limited to the following languages: PHP, Java, Python, and Go. The two initial web frameworks we decided to use were Ruby on Rails and Flask because members of our team were familiar with them. We eventually decided on Flask because of familiarity and that Google App Engine can not run Ruby applications.
2. Choosing a hosting provider
    a. Our application needs to be publicly accessible from the Internet. In order to do that it needs to be hosted on some platform. Our three options when choosing a hosting provider were Heroku, Google App Engine, a Raspberry Pi cluster, and rolling our own server on a Virtual Private Server. We eventually chose Google App Engine because of the Python SDK it offers. We also chose Google App Engine

because of their NDB Datastore which allows applications to easily store and access information without rolling their own SQL or SQLite server. Doing this allows us to not be worried with the intricacies of SQL and focus on developing the application. Google App Engine also offers a free subdomain from their appspot domain, so we do not have to purchase a domain to access the server with a distinct URL, but we still have the option for a custom domain in the future.

3. Thin client vs Fat client
    a. Given that we are using a client-server architecture, we needed to decide whether our client (the Android application) would be a thin client or fat client. A thin client allows more processing to be done on the server, but means that a lot of data will have to flow between the client and the server. While a fat client does a lot more processing and allows for less data transfer between client and server, but allows for more failure points to occur on the app (which makes error checking harder) and it would also lead to a more bloated application due to the added features in the application.
4. Using Client Server Architecture
    a. A client-server architecture is the most popular architecture, and instead we could have chosen a peer to peer architecture. However, Flask and Google App Engine only have support for the client-server architecture. There are also many libraries on Android that support the client-server architecture therefore the obvious choice was to use the client-server architecture.
5. Choosing how to serialize data
    a. When sending data back and forth between the server and client, it is important to have an agreed upon format that the server and client understand. The language should be able to serialize the data of the server/client and transmit them to each other. The two most common ways of serializing data to send between clients and servers are XML and JSON. XML is much older and there are many libraries to help with the creation of serialized XML, but JSON is a much cleaner, more modern way of doing so. Creating and sending JSON is a very simple process in both Python and Android, therefore we decided to use JSON.
6. Choosing what to cache, how and for how long
    a. If we cache too much, then using cached data will be nearly as inefficient as requesting that data from the server. We will cache all of a user's favorited maps on their devices until they refresh the list. We

can achieve this goal with a refresh button and by storing JSON data in their app directory. The server will also cache using the built in caching associated with NDB.

Design Details:

- Main UI Activity

In Android, the UI for an application is drawn from an activity, which is just a class in and of itself. It will manage drawing the UI and initiate the routines which download data from the server.

- Map

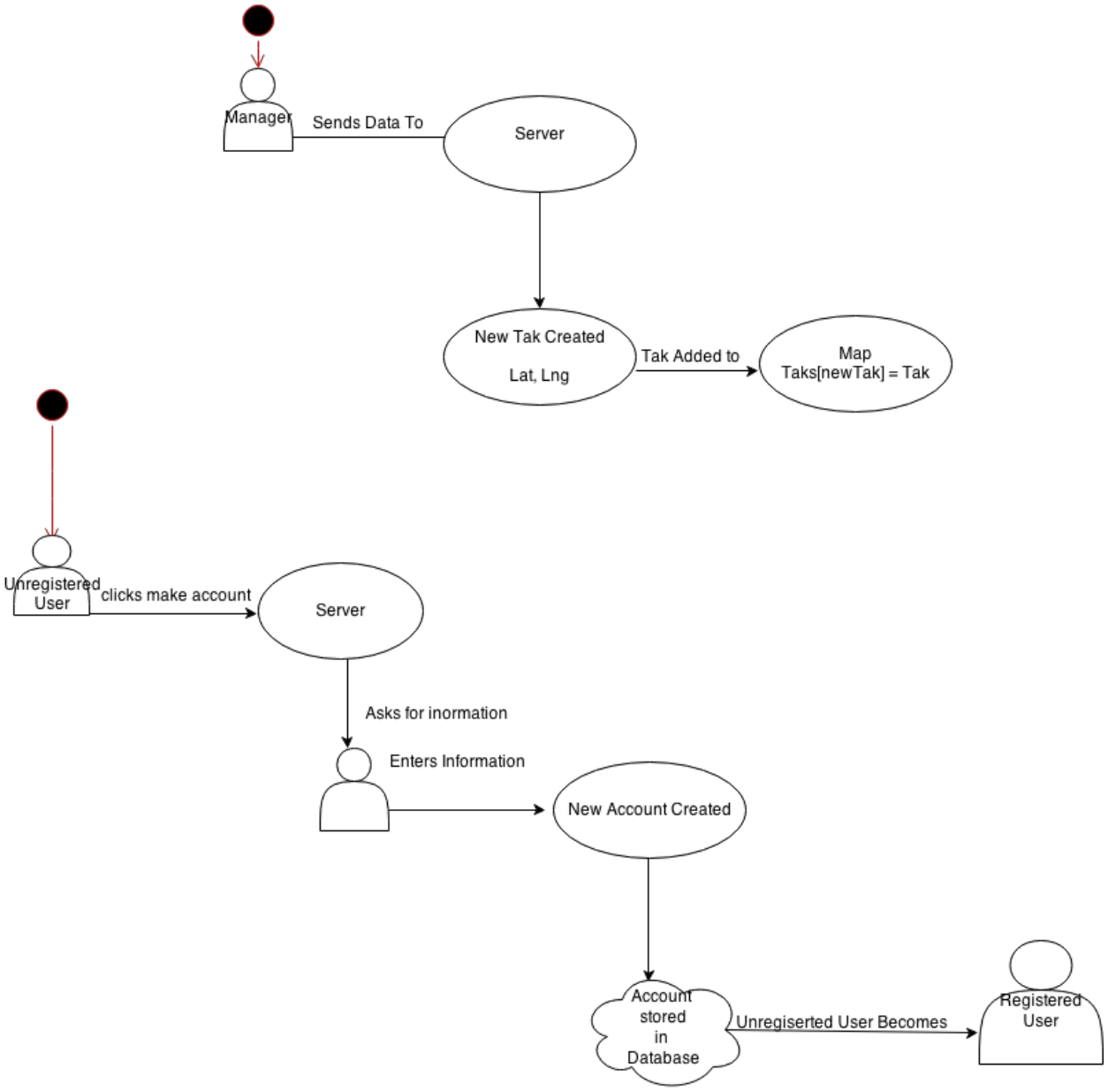
The Map class will encapsulate all the information pertaining to a single map the user wants to display, including a list of Taks, the map name, and information about who can access the map. If constraints allow for it, maps will also contain meta-data.

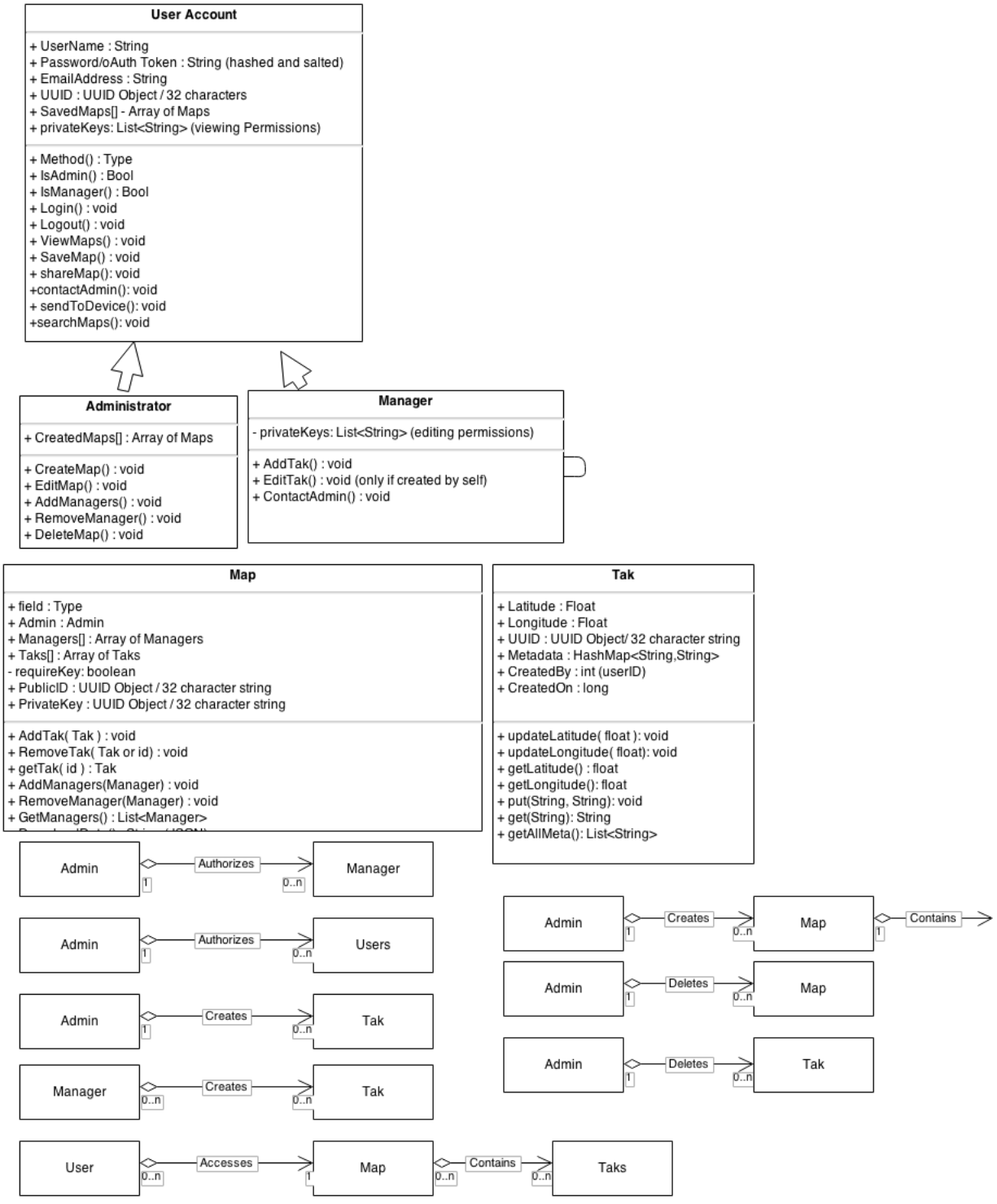
- Tak

A Tak will contain a label, a latitude/longitude point, and a dictionary/hashmap which contains all of the additional metadata the user has decided to store connected to the single Tak. Every Tak will exist as a member of a list contained inside a Map object.

- Account

An account can be either a user, manager, or admin of a map. Users will be able to create maps, edit maps(if they have permission), and view maps. Creators of a map(admins) will be able to give permission to certain users, who will then be able to add/edit/remove taks on a map.


Manager
Sends Data To
Server
New Tak Created
Lat, Lng
Tak Added to
Map
Taks[newTak] = Tak
Unregistered
User
clicks make account
Server
Asks for inormation
Enters Information
New Account Created
Account
stored
in
Database
Unregiserted User Becomes
Registered
User


User Account
+ UserName : String
+ Password/oAuth Token : String (hashed and salted)
+ EmailAddress : String
+ UUID : UUID Object / 32 characters
+ SavedMaps[] - Array of Maps
+ privateKeys: List<String> (viewing Permissions)
+ Method() : Type
+ IsAdmin() : Bool
+ IsManager() : Bool
+ Login() : void
+ Logout() : void
+ ViewMaps() : void
+ SaveMap() : void
+ shareMap(): void
+contactAdmin(): void
+ sendToDevice(): void
+searchMaps(): void
Administrator
+ CreatedMaps[] : Array of Maps
+ CreateMap() : void
+ EditMap() : void
+ AddManagers() : void
+ RemoveManager() : void
+ DeleteMap() : void
Manager
- privateKeys: List<String> (editing permissions)
+ AddTak() : void
+ EditTak() : void (only if created by self)
+ ContactAdmin() : void
Map
+ field : Type
+ Admin : Admin
+ Managers[] : Array of Managers
+ Taks[] : Array of Taks
- requireKey: boolean
+ PublicID : UUID Object / 32 character string
+ PrivateKey : UUID Object / 32 character string
+ AddTak(Tak) : void
+ RemoveTak(Tak or id) : void
+ getTak(id) : Tak
+ AddManagers(Manager) : void
+ RemoveManager(Manager) : void
+ GetManagers() : List<Manager>
Tak
+ Latitude : Float
+ Longitude : Float
+ UUID : UUID Object/ 32 character string
+ Metadata : HashMap<String,String>
+ CreatedBy : int (userID)
+ CreatedOn : long
+ updateLatitude(float): void
+ updateLongitude(float): void
+ getLatitude() : float
+ getLongitude(): float
+ put(String, String): void
+ get(String): String
+ getAllMeta(): List<String>
Admin
Authorizes
Manager
1
0..n
Admin
Authorizes
Users
Admin
Creates
Tak
Manager
Creates
Tak
User
Accesses
Map
Contains
Taks
Admin
Creates
Map
Contains
Admin
Deletes
Map
Admin
Deletes
Tak