Design Document for @10dance

Group#B4

Curtis Ullerich

Brandon Maxwell

Todd Wegter

Yifei Zhu

Version	Date	Author	Change
0.1	10/08/04	SM	Initial Document
0.2	03/31/12	TW	Skeleton and First Issue Added
0.3	04/04/12	CU	Module Interfaces Added
0.4	04/04/12	TW	Mobile App Process and 2nd and 3rd Issues Added
0.5	04/06/12	YZ	4th Issue Added
0.5	04/06/12	BM	Server Processes Added
1.0	04/06/12	TW	Edited for Final Submission

Table of Contents

1 Introduction

[NONE]

1.1 PURPOSE

[NONE]

1.2 SCOPE

[NONE]

1.3 DEFINITIONS, ACRONYMS, ABBREVIATIONS

Term	Description
[NONE]	

1.4 DESIGN GOALS

[NONE]

2 References

[NONE]

3 Decomposition Description

3.1 MODULE DECOMPOSITION

[NONE]

3.1.1 [NONE]

3.2 CONCURRENT PROCESS

[NONE]

3.2.1 [NONE]

[NONE]

3.3 DATA DECOMPOSITION

3.3.1 [NONE]

3.4 STATES

3.4.1 [NONE]

4 Dependency Description

4.1 INTERMODULE DEPENDENCIES

[NONE]

4.2 INTERPROCESS DEPENDENCIES

[NONE]

4.3 DATA DEPENDENCIES

[NONE]

5 Interface Description

5.1 MODULE INTERFACES

a) Google App Engine Datastore

b) http:// Communication Between Mobile Application and Server

c) Mobile Application html5 localStorage

d) JSP-Backend Interface

5.1.1 Google App Engine Datastore

This sub-system is our data storage location for all user data, including account information, and most importantly, attendance data. GAE's (Google App Engine's) datastore does not allow fields of objects to be objects (other than String and wrapper classes for primitives) and does not play well with polymorphism, which requires us to flatten our class hierarchy as defined in our original UML diagram. Our desired schema is a top-level Person class with fields for all personal information as Strings and an AttendanceRecord, which in turn has fields for all Tardies, Absences, AbsenceRequests, ClassConflictForms, etc. The datastore also contains a top-level Event object that has child types of Performance and Absence. We have rewritten this to avoid polymorphism (having just a Performance and Absence without a common superclass and having a User object instead of Student, TA, and Director objects with a common parent).

In order to maintain a rich User object and not be required to serialize all data into Strings, we are using an interface to define getting and setting objects relative to a User. Each User has getter and setter methods for all account information (netID, first name, last name, etc) and for its respective AttendanceRecord.

Example:

public void setAttendanceRecord(AttendanceRecord record);
This method sets the value of a private String field of the User to the key of AttendanceRecord record, which is stored as a top-level object in the datastore.

public AttendanceRecord getAttendanceRecord();
This queries the database using the private String field of the User that corresponds to the User's Attendance Record.

We add and remove entities from the database using an interface defined in our DatabaseUtil class. Simple getters and setters encapsulate the retrieval of the database service and persistence.

public static User getUser(String netID)
public static void addUser(User guy)

public static void addEvent(Event toAdd)
public static String[] listAllUsers()

5.1.2 http:// Communication Between Mobile Application and Server

This is accomplished through a data serialization interface. The mobile app needs to send all relevant data currently stored in localStorage to the server. It concatenates the keys of all absences, tardies, and events stored in localStorage (both as described above) into a comma-delimited string that is sent to the server via an HTML POST request and parsed using a servlet's doPost method. This interface for serialization makes parsing easy, as two calls to String.split(char delimiter) can create a 2D array of everything that was previously stored in localStorage at the client side.

The mobile app also needs to be able to pull the most recent list of students from the server. This is handled in a single JSP, so it does not require an interface of its own, but instead uses the HTML5 localStorage interface defined below.

5.1.3 Mobile Application html5 localStorage

HTML5 localStorage is a key-value map that deals only with Strings. Because of the many types of data we store in localStorage, we defined an interface for the common serialization of four different data types: Users, events, tardies, and absences.

Each *key* is a space-delimited string of the form "prepend firstname lastname netID date startTime endTime rank". Prepend is an identifier that determines the type of value stored. Firstname, lastname, and netID are strings that describe the user information. Date is of the format YYYY-MM-DD. StartTime and endTime are in 24-hour time in the format HHMM. Rank is a 6 digit number. The *value* is the desired string to be printed when displayed to the user.

Absences use this format explicitly. Tardies replace the endTime with a pipe "|" signifying null. Events have pipes in place of all unneeded values (namely, firstname, lastname, netID, and rank). Users have pipes for date, starttime, and endtime.

Example for an absence:
"absentStudentRehearsal Curtis Ullerich curtisu 2012-3-28 1630 1750 000001"

This allows for util functions for storing and retrieving items from localstorage, as well as searching and aggregating entities of a particular type for sorting, storing, POSTing, etc.

5.1.4 JSP-Backend

The JSPs that comprise the GUI for our web application pull user information from the datastore using the interface defined in 5.1.1 above. We use a standard interface for servlets for pushing data to the server using an HTML POST request:

public void doPost(HttpServletRequest req, HttpServletResponse resp)

For each type of servlet necessary (in which we send various types of data and handle the POST differently), we use this method to accept data. Each servlet class extends HttpServlet.

From the HTML page, a POST is done as the method of a standard HTML form and sends the values of all form fields to the servlet, where they are processed in our doPost method.

5.2 PROCESS INTERFACES

a) Mobile Application

a) Mobile Application

b) Server Threads

5.2.1 Mobile Application

The mobile application is written as an html5 website with appcaching. This allows the app to work much like a website, but it can be viewed offline after being viewed online. Users start the mobile app by simply navigating to it in their computer's web browser. The user can then interact with the mobile app through their web browser as they would like. The mobile app will run in the user's web browser until it is closed. When the mobile app sends data to the server or asks for data from the server, the two applications send data between each other over http://. The server can initialize an instance of the mobile app as data transfer is always initiated by the mobile app. The mobile app also never initializes the server as the server does not shut off.

5.2.2 Server Threads

The server threads are handled entirely by Google App Engine. There are two different types of processes that Google App Engine contains. The first is the

thread dispatcher and the second type is a “main” thread for the client. When a client connects to the website the thread despatcher creates a Thread for the client and launches the related processes. When the client is finished on the website, Google App Engine terminates those main threads.

6 Detailed Design

[NONE]

7 Design Rationale

7.1 DESIGN ISSUES

Several issues were encountered during the course of our project. One of the biggest issues was deciding how to host the system. Determining how to implement our mobile application was also a large implementation decision. We also encountered issues when designing the format for storing users in the datastore. The method of storing items to the datastore also proved problematic.

7.2 SYSTEM HOSTING

7.2.1 Description

We needed to find an appropriate method for hosting our application. Special consideration was given to this decision as our project will be used by real clients starting this fall.

7.2.2 Factors affecting Issue

We were primarily concerned with how the clients would be able to access and maintain the server and the cost to the clients.

7.2.3 Alternatives and their pros and cons

Two different approaches were considered. First, we looked at hosting a local server using Apache. This system would be fairly easy to implement, and it is the standard approach taken in this class. However, it would require finding space to run and host the server. This would likely require a sizeable investment on the part of our clients.

We also considered using the Google App Engine. This system hosts and runs both the server and mobile applications in the cloud at no cost. The Google App Engine also provided an integrated database, or datastore, so that a separate database would not need to be implemented. Unfortunately, none of our members had any experience with this system, so its implementation could be more difficult.

7.2.4 Resolution of Issue

The Google App Engine was selected for our project, primarily due to its cost of implementation. We decided that it would be best not to require our clients to maintain and pay for their own server. The integrated datastore has also proved to be a great benefit of using the Google App Engine.

7.3 MOBILE APPLICATION IMPLEMENTATION

7.3.1 Description

The decision for how to write our mobile application was not a clear-cut decision. We needed to find a way to make an application that would be cross platform, interface with the server, and be accessible offline.

7.3.2 Factors affecting Issue

We wanted to find a language that would be fairly simple and run on any sort of platform the marching band could use on the field. Ideally, the marching band will be running tablets, so we needed an interface that would touch friendly. The mobile app also needed to be able to interface with our server, yet still be available offline since the practice fields are outside of the ISU wireless network.

7.3.3 Alternatives and their pros and cons

We considered using Java to write a real application and html5 to write a web app that would be available offline. We were more familiar with Java, so this would have been much easier to write. Also, the server's logic is all written in Java, so common code could have been used for communicating between the two applications. Unfortunately, Java apps are not supported by Apple iOS products, and we wanted the mobile app to be accessed from iPhones or iPads if necessary.

Html5 would allow us to write a web app that would be available offline through appcaching. We had not written html5 before, so this method would be more difficult to write. We also weren't sure how we would be able to communicate between a Java server and html5 application. However, this application could run on any device with a web browser. It wouldn't even need the internet to run the app if the web app pages were stored locally.

7.3.4 Resolution of Issue

We decided to write our mobile app in html5 to allow it to be used on any device. It also turned out to be very easy to interface html5 with the server, and the mobile app logic was all done in javascript, which was similar to Java and easy to learn. Html5 allowed us to create a mobile app that would work on any device with a web browser, and app-caching allows the web pages to be stored in the user's web browser so that the app can be access without an internet connection.

7.4 USER CLASS IMPLEMENTATION

7.4.1 Description

We needed to find a solution to store our Student, TA, and Director java classes. Each of these java classes must be able to get information from our datastore.

7.4.2 Factors affecting Issue

In order to get or store a user's information, we created a super class "Person.java" which contains all common user information, and three sub which classes extended from Person, Student, TA, and Director. We were hoping to connect the Google App Engine datastore and our java classes. Unfortunately, the Google app engine database does not play nicely with polymorphism. For example, we cannot store Student instances in the datastore because they would sometimes be stored as a Student and other times as a Person. This makes retrieving all the stored Students impossible. We could not debug this issue.

7.4.3 Alternatives and their pros and cons

We could choose another system to store our database and keep our user inheritance relationships. Pros: we would be able to use java inheritance, and our java code would be more organized, enabling us to do more code reuse. Cons: it is almost the end of semester, and if we change our method of data storage, the learning curve and potential bugs would push us past the deadline.

Conversely, we could remove the inheritance relationship in our java class hierarchy. Pros: this will solve the database storage problem. The java code will be able to easily connect to the Google app engine data storage. Cons: we lose the inheritance in our java code. This will create code duplication, and might be hard to maintain if we want to delete some functionally from all the classes. It is also harder to keep track of bugs.

7.4.4 Resolution of Issue

We decided to delete the inheritance relationship in our java class hierarchy, deleting all of our super classes and adding all of the common methods to all the sub-classes. We can now store and get information from the Google app engine database. In the short term, this was a faster solution to the issue than learning a whole new database. Unfortunately, in the long run we will spend more time maintaining our code and fixing common bugs in of our java classes.

7.5 DATASTORE IMPLEMENTATION

7.5.1 Description

We needed a method of storing data locally in the mobile app while no internet connection was present so it could be saved until the internet was reconnected and the data could be uploaded to the server. We used the localStorage provided by html5 as this temporary storage, but this had some limitations.

7.5.2 Factors affecting Issue

LocalStorage only allows the storing of strings in a key based map. This made it difficult to get the desired information from a specific entry. We needed a way to serialize the information we wanted to store and retrieve that serialized data.

7.5.3 Alternatives and their pros and cons

We created a custom serialization interface that was able to store and retrieve all the information we need through the localStorage. The biggest issue we faced in this process was whether to store the information in the key or the value of each entry in the map. The key would be easier to search, but the value would be slightly more secure. The information for each entry would also create a unique identifier that would be easy to request because its form would be known and easy to build. Using the key for the important data would also allow us to store a more nicely formatted string in the value for printing or some other piece of unique information for an entry. Unfortunately, neither of these methods are easily extensible, so realizing that a piece of information has been omitted from the key format can be costly to correct.

7.5.4 Resolution of Issue

The key was used to store all the information because the benefits of this approach greatly outweighed the risks. So far, this interface has proved easy to use and secure, and it has given us zero issues.

8 Traceability

No	Use Case/ Non-functional Description	Subsystem/Module/classes that handles it
1	[NONE]	[NONE]
2	[NONE]	[NONE]