Potlatch Design Documentation

Mark H. Butler (markhenrybutler@gmail.com)
Sunday 19th October 2014

1. Introduction

Potlatch is a way for *users* to share *gifts* consisting of a *title*, an optional *description*, an *image*, and in the case of this implementation, *an optional video*. Gifts are organized into *gift chains* that consist of one or more related gifts. Users can indicate if they were *touched* by a gift and *flag* inappropriate or obscene gifts.

In this report we will review the design of the Potlatch project. First we will explicitly discuss some design issues highlighted in the requirements document. Then we will outline the design of the client application. Then the we will outline the design of the server application. Then we present the original wire frames that were created for the initial design. Then we include annotated screenshots of the final application. This makes it possible to see how the application design evolved during implementation.

2 Issues to be discussed from the requirements document

The requirements document highlighted a number of design issues for consideration:

1. How will the gift data be stored? In your device? In the remote service?

Data is stored in a remote service as specified in the grading rubric.

2. What will the user interface screens look like? How will the user navigate between the different screens?

Full details of this and wire frames are in a subsequent section.

3. How and where will users attach Gifts to a Gift Chain?

Users attach gifts to gift chains at creation time by specifying the gift chain name. There is an autocomplete text box which helps them pick gift chains that already exist.

4. How will users delete Gifts from a Gift Chain, as well as delete Gift Chains altogether?

A user cannot delete a gift from gift chain, they can only delete it. This removes it from that gift chain. Gift chains cannot be deleted. However if now gifts are associated with a gift chain, it will not be visible, except in the auto complete box when the user types the name of that gift chain while creating a new gift.

5. How, when and how often will the user enter their user account information? For example will the user enter this information each time they run the app? Will they specify the information as part of a preference screen?

The username and password are entered by the user in the login screen. This information is then stored in the shared preferences along with some other information about queries ETC that allows the application between the different activities. The client then uses this username / password to obtain a token from the service that is uses in subsequent API calls.

6. How will gifts and gift chains be accessed? Will there be some initial default display? Will users have to enter search criteria? If so, how will the user get access to the search interface?

Gift chains are accessed from a “link” icon of a gift. Currently the initial default display is all gifts. Clearly this will not scale. Users can enter search criteria for title via a search bar at the top of the application. They can also select different query modes e.g. gifts by current user, gifts by a specific user and top gift givers.

7. Will all search results be displayed at the same time? Or will there be some initial default display? Will user have to enter a search criteria? If so, how will the user get access to the search interface?

Yes, unfortunately there is no paging in the current implementation so the full set of results are returned. Clearly such an approach will not scale. As already mentioned, there is a search bar at the top that allows users to specify a text query or a query type.

8. How will users indicate that they were touched by a Gift? Can they undo their decisions? How will they flag inappropriate or obscene Gifts?

They indicate they were touched by pressing the heart icon, or they flag gifts by pressing the flag icon. They undo their decision by pressing the icon a second time.

9. What preferences can the user set? How will the app be informed of changes to these user preferences?

They can specify update frequency and also whether to hide flagged gifts. This is demonstrated in the video.

10. How will users navigate between viewing / searching for Gifts and viewing top Gift givers?

They press the "top gift givers" button (indicated by a trophy), or the "all" button (indicated by a number of users) or the "me" button (indicated by a single user) in the action bar to cycle through the different views.

11. How will the app handle concurrency issues, such as how periodic updates occur - via server push or app pull? How will search queries and results be efficiently processed? Will the data be pulled from the server in multiple requests or all at one time? Will the server data include full-sized images or thumbnail plus URLS pointing to full-sized images?

Periodic update is done by app pull. In general concurrency issues are handled using the **RxJava [RxJava]** library, as this is supported by **Retrofit [Retrofit]**, the library used to create the client REST interface. When a search query is sent to the server, it is sent on a different thread, with a callback to the UI thread. Then when the call completes, it triggers a refresh of the gift display. The periodic updates work in a similar manner.

The app reduces the size of the image, adjusts the orientation according to the supplied **Exif** information, then compresses it to PNG format before uploading it to reduce the size of the images stored on the server and reduce the upload / download time. It does generate thumbnail images for videos but not for images.

12. Will the app cache information on the local device e.g. in a ContentProvider?

The app uses the **Picasso library [Picasso]** from Square to cache images. This speeds up image download by caching data to memory or disk.

30. Will the app provide extensions and improvements that go beyond the minimum requirements? For example if an app collects location information for each Gift, queries by location and then uses Maps to display Gifts at their locations, then it may be necessary to modify the various app databases and query facilities. Also, using Maps may require students to have access to an actual device. Also, how will these enhancements affect the rest of the app?

The intention was to support uploading and downloading of videos although there was not time to test this functionality (so it may not work).

Work on the application did involve using some new libraries / tools that I had not encountered before, some of which were presented in class but a few others that had not. Because new tools were presented in class, this encouraged me to investigate related tools.

Specifically **Retrofit** supports a concurrency framework called **RxJava** so that was used. **Picasso** was used because it was developed by Square, the same company who produced Retrofit and **Butterknife**. **Gradle**, the build system Jules White presented, uses **Groovy**, so behaviour driven testing was implemented using **Groovy** and **Spock**, a Groovy **BDD (Behaviour Driven Design)** framework. Finally **Butterknife** and **Spring** both demonstrate how annotations can be used to improve Java to lose some of its verbosity compared to more modern languages such as **Scala**, so I decided to use **Lombok**, another Java annotation framework. These tools will be discussed in more depth in a later section. All these tools were new to me and this is the first time I had used them.

3 Client Application Design

The client application will be implemented as Android Java application. It should allows users to:

- Search for gifts with containing a specific sub-string in the title.
- Browse gifts and gift chains.
- Browse top gift givers i.e. users whose gifts have touched many people.
- Browse gifts submitted by the current user.
- Create new gifts.
- Create new gift chains.
- Add gifts to new and existing gift chains.
- Like gifts that have touched them.
- Flag gifts as being inappropriate or obscene.

The app will go beyond the minimum specifications by providing support for gifts that optionally contain videos as well as images. This requires the use of the multimedia capture and multimedia playback APIs.

In the first version of the design document, the client application is described using a number of wireframe diagrams. The UI for the client has changed so they have been replaced with annotated screenshots enclosed at the end of this report.

The login process will the password grant flow in OAuth2. The server application will provide an authentication server. The user will enter their username and password into the client, and it will then send that information via HTTPS to the authentication server in order to get an OAuth token. This token has a finite lifetime. The username and token will be stored securely on the client and will not be accessible by other applications.

3.1 Browsing

There are five types of views returning search results:

- *All gift view* displays all the gifts.
- *Title query view* displays all the gifts whose title match a specific query if a search query has been entered. The query is entered in the search box at the top of the page.
- *Gift chain view* display the gifts associated with a specific gift chain.
- *Top gift giver view* display the users who received the most number of votes for posting touching gifts .
- *User view* displays the gifts created by the user.

The query views, gift chain views and my gifts views can be displayed either as a list, displaying the image, title and associated text, or as a grid, only displaying the image. The top gift giver view can only be displayed as a list. By default, the application returns the query list view.

The results are ordered either by the number of votes they have received (highest first) or by when they were published (most recent first).

3.2 Interaction with gifts

If users are logged in, then they can vote for gifts that touched them and they can flag gifts as being inappropriate or obscene. If they vote for a gift or flag it, then the icon associated with that action becomes highlighted. To remove their vote or unflag a gift, they simply press the icon a second time and it will return to its normal colour.

3.3 User preferences

The user preference page allows users to select whether they want results being displayed ordered by the number of votes they have received or by order of creation. They can also select whether inappropriate or obscene content should be hidden.

3.4 Updating data

Updating touch votes at a specific frequency is quite difficult. One way to update the client is for the client to poll the server at a certain interval e.g. every minute, five minutes or every sixty minutes. However this creates additional load for the server, so in general the preferred way to do this is to use Android push notifications. Here if the state changes at the server, then the server sends a message to the client via an Android push notification. When the client receives the notification, it polls the server. This "push" approach could be used to notify an email application of the arrival of a new message, or on auction site to tell the user when they have been outbid because it is clear what the user is interested in.

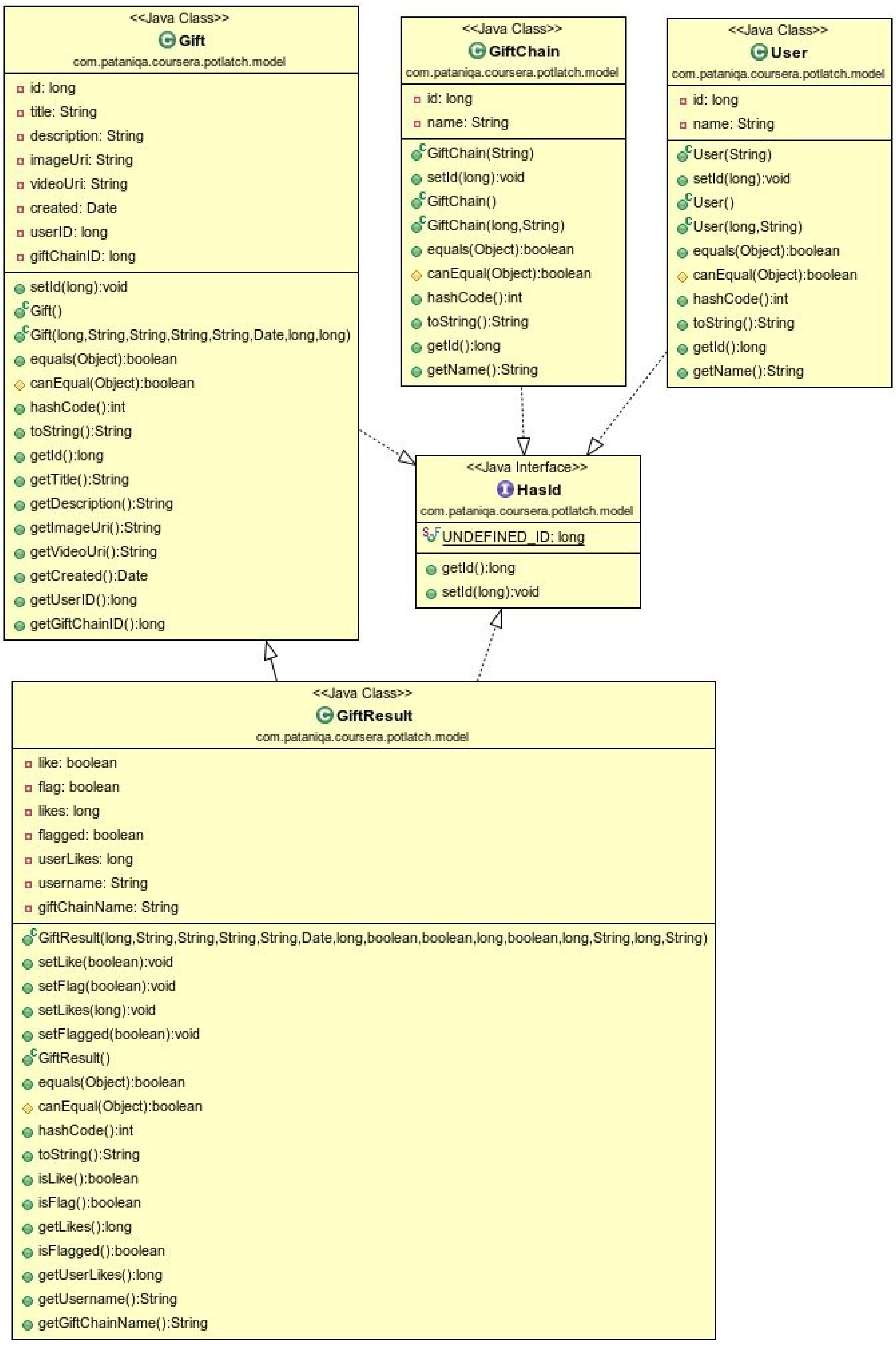
However here the problem is how does the server determine what the user is interested in? Is it the gifts that they themselves have created? Is it the gifts that they have voted as touching them? Is it any gift that they happen to be viewing at that particular moment? Because in this application the focus of the user is changing all the time, using push notifications is not straightforward. Also, arguably, if the user is not actually using the application, getting these notifications might be undesirable as they are not that important to the user so they may only serve to reduce the battery life of the device. The user might only be interested in updates when they are using the application. For example, Facebook can generate a large number of email notifications, or on a mobile device tray notifications. However not all users feel such notifications add value, and some more feel they are an irritation.

Therefore here we propose that the initial version of the application will only support manual update of the number of touched votes. Once that version of the application works successfully, if there is time available, then the application will be extended to use updating via app pull. In this first version if the Potlatch application is active, the client will poll this API at the configured interval. So we accept will that this approach may not be as efficient as server push, but we prefer it because it is easier for the client to determine what updates it requires if any rather server.

3.5 Data model

On the client, there four main classes in the data model: Gift, User, GiftChain and GiftResult. GiftResult is a denormalized view of the database used to populate the user interface. All four of these classes inherit from the HasId interface which indicates that they have a gettable / settable ID i.e. they are database records.

Here is an outline of the main classes used in the data model :


<<Java Class>>
Gift
com.pataniqa.coursera.potlatch.model
id: long
title: String
description: String
imageUri: String
videoUri: String
created: Date
userID: long
giftChainID: long
setId(long):void
Gift()
Gift(long,String,String,String,String,Date,long,long)
equals(Object):boolean
canEqual(Object):boolean
hashCode():int
toString():String
getId():long
getTitle():String
getDescription():String
getImageUri():String
getVideoUri():String
getCreated():Date
getUserID():long
getGiftChainID():long
<<Java Class>>
GiftChain
com.pataniqa.coursera.potlatch.model
id: long
name: String
GiftChain(String)
setId(long):void
GiftChain()
GiftChain(long,String)
equals(Object):boolean
canEqual(Object):boolean
hashCode():int
toString():String
getId():long
getName():String
<<Java Class>>
User
com.pataniqa.coursera.potlatch.model
id: long
name: String
User(String)
setId(long):void
User()
User(long,String)
equals(Object):boolean
canEqual(Object):boolean
hashCode():int
toString():String
getId():long
getName():String
<<Java Interface>>
HasId
com.pataniqa.coursera.potlatch.model
UNDEFINED_ID: long
getId():long
setId(long):void
<<Java Class>>
GiftResult
com.pataniqa.coursera.potlatch.model
like: boolean
flag: boolean
likes: long
flagged: boolean
userLikes: long
username: String
giftChainName: String
GiftResult(long,String,String,String,String,Date,long,boolean,boolean,long,boolean,long,String,long,String)
setLike(boolean):void
setFlag(boolean):void
setLikes(long):void
setFlagged(boolean):void
GiftResult()
equals(Object):boolean
canEqual(Object):boolean
hashCode():int
toString():String
isLike():boolean
isFlag():boolean
getLikes():long
isFlagged():boolean
getUserLikes():long
getUsername():String
getGiftChainName():String

4 Server application

The server application will be written in Java using Spring. The client and the server applications communicate via HTTP or HTTPS when required, and the server provides a REST based interface that is queried by the client. The server supports multiple users via individual user accounts.

The gift data will be stored on the remote service. If the gift contains a video, then that will only be retrieved if the user presses the image in the detail view of the gift to initiating playing the video.

4.1 REST API design

This section defines the REST API that the server will use for communicating with the client:

Users

GET /user	Get all users.
GET /user/{id}	Get a specific user.
POST /user	Create a new user.
PUT /user/{id}	Update a user.
DELETE /user/{id}	Delete a user.

Gift chains

POST /giftchain	Create a gift chain.
GET /giftchain	Get all the gift chains.
GET /giftchain/{id}	Get a specific gift chain.
PUT /giftchain/{id}	Update a gift chain.
DELETE /giftchain{id}	Delete a gift chain.

Gifts

GET /gift	Get all the gifts.
POST /gift	Create a gift.
PUT /gift/{id}	Update a gift.
DELETE /gift/{id}	Delete a gift.
GET /gift/{id}	Get a specific gift.
PUT /gift/{id}/like/{like}	Update a gift as being liked / unliked by a specific user.
PUT /gift/{id}/flag/{flag}	Update a gift as being flagged / unflagged by a specific user.

*GET /gift/title?title={title}
&order={order}
&direction={direction}
&hideflagged={hide}*
	Query gifts by title, specifying the order, order direction and whether to hide flagged results.

*GET /gift/user?title={title}
&user={userID}
&order={order}
&direction={direction}
&hideflagged={hide}*
	Query gifts by user, specifying a title query, order, order direction and whether to hide flagged results.

*GET /gift/topGivers?title={title}
&user={userID}
&direction={direction}
&hideflagged={hide}*
	Query by top gift givers, specifying a title query, order, order direction and whether to hide flagged results.

*GET /gift/giftchain?giftchain ={giftchain}
&order={order}
&title={title}
&user={userID}
&direction={direction}
&hideflagged={hide}*
	Query by gift chain, specifying a title query, order, order direction and whether to hide flagged results.

POST /gift/{id}/video	Save a gift video on the server.
GET /gift/{id}video	Get a gift video from the server.
POST /gift/{id}/image	Save a gift image on the server.
GET /gift/{id}/image	Get a gift image from the server.

5 Implementation

Note my project does include code from other sources. Specifically it uses example code presented by Jules White and it refers to a few Stack Overflow answers. I think this is normal in development. But due to the honour code I have clearly indicated in the source code where this is the case with a citation of where I got the material.

The code consists of three modules: **PotlatchCommon**, **PotlatchClient** and **PotlatchServer**.

PotlatchCommon contains all the code common to the client and the server, specifically:

– the common data model (Gift, GiftResult, GiftChain and User)

– a **facade** separating the data store from the client,

– a **Retrofit client [Retrofit]** that connects to a remote data store and implements the facade.

The reason the facade was created was to speed up development time, as a local store was created so that the client could run without a separate server. This local store implemented the facade using **SQLlite**. The local store implementation is still in the PotlatchClient. This can be seen in

```
PotlatchClient/src/com/pataniqa/coursera/potlatch/store/local/
```

Because Retrofit supports **RxJava [RxJava]**, a reactive extension framework for multithreading / concurrency, I decided to use that framework. Therefore the facade uses RxJava **observables** because the operations it implements are require network access and are potentially long running. It was also necessary to use a custom HTTP client as the server certificate is self-signed, so clients will not connect to it via HTTP. Because Retrofit and another library used in the client, **Picasso,** use **OkHttp [OkHttp]** also developed by Square this was chosen as the basis of the HTTP client.

OkHttp uses an I/O library called OkIo [OkIo]. These components are used in the Retrofit client.

PotlatchClient incorporates the common code, then it contains the local store implementation (which is no longer used) and the user interface.

PotlatchServer contain three components:

- The authorization code used to authenticate user / passwords.
- The server which contains the application and the controllers for the Gift, Gift Chain and User REST services.
- A data model that maps the common data model to a database using Spring JPA annotations.
- A repository directory that contains repositories for storing gifts, gift chains, users and gift metadata (who liked or flagged different gifts). These repositories leverage Spring data to implement create-retrieve-update-delete and query.

All the modules use Project **Lombok [Project Lombok]**, which is an annotation framework for Java that allows annotations to replace boilerplate such as getters, setters, toString, hashcode and equals.

As already mentioned the Android client also uses **Picasso [Picasso]**, a library to simplify displaying and caching images on Android clients, and **Butterknife [Butterknife]**, a library described on the course by Jules White, that uses annotations to reduce common code in Android applications, in a similar way to Lombok.

The server uses **Spring Boot [Spring Boot]** to provide server functionality, **Spring Data REST [Spring Data REST]** to perform object relational mapping and **Spring Security Oauth [Spring Security Oauth]** to provide security as described in the Cloud Computing course.

Testing the server is done using **Groovy [Groovy]**, the language used to create **Gradle [Gradle]** and using **Spock [Spock]**, a BDD testing framework written in Groovy.

For more information on compiling the code and working with the code in an IDE, see the README.md file included with the code.

6 References

[Butterknife] http://jakewharton.github.io/butterknife/

[Gradle] http://www.gradle.org/

[Groovy] http://groovy.codehaus.org/

[Picasso] http://square.github.io/picasso/

[Project Lombok] http://projectlombok.org/

[OkHttp] http://square.github.io/okhttp/

[OkIo] http://square.github.io/okio/

[Retrofit] http://square.github.io/retrofit/

[Rxjava] https://github.com/ReactiveX/RxJava

[Spring Boot] http://projects.spring.io/spring-boot/

[Spring Data REST] http://projects.spring.io/spring-data-rest/

[Spring Security Oauth] http://projects.spring.io/spring-security-oauth/

[Spock] https://code.google.com/p/spock/