SHOPNOTE SYNCHRONISATION DOCUMENTATION

This document shows how the server works and how you can interact with it.

INTRODUCTION

The idea of this server implementation ist, that the server shouldn't care about what the user is changing on his list. He just forwards all changes to the users which also participate in this list. He manages only the connection between those users (the forwarding).

1 GENERAL TECHNOLOGIES

1.1 COMMUNICATION

All communication between the server and the client is done over sockets. The information exchange is always started by the user and never by the server. That means the user needs to open and connect the socket to the server.

To learn more about sockets see: http://docs.oracle.com/javase/tutorial/networking/sockets/

1.2 STORAGE

There are two different type of storage solutions used to persist user data on the server

- SQLite:
  - Stores all information the server needs to forward changes to the right client
  - This includes phonenumbers, userIds, which list is shared with which users etc.
- Neodatis object database
  - Stores all changes of lists. The server doesn't care about those changes, he just forwards them to the user.
  - This includes: Shared list creation, Item adding, Item removing, Item buying etc.

Notation: We call the information of the...
...SQLite database server class information
...Neodatis object database client class information

2 JAVA CLASSES

2.1 COMMUNICATION

2.1.1 MANAGER

All the communication on the client side is managed by the SyncManager object. It's method synchronize() opens a Socket and sends all pending changes to the server. This method also gets new changes from the server.

There are a few Helper classes for the Syncmanager which handle single tasks of the stuff mentioned above:

- RequestSender: handles the technical side of the communication. This is done on a new thread (not the UI thread) to speed up the interaction
- RequestQueue: Wrapper around an ArrayList to keep the requests that have not been synched
- AnswerHandler: Processes all changes comming from the server

2.1.2 COMMUNICATION CLASS (REQUEST)

The abstract class Request defines the standard object that is passed over the connection. It contains a unique identifier for each user (phonenumber).

There are several subtypes of Request:

Name	Purpose	Information class
EmptyRequest	-	-
RegisterRequest	Register yourself on the server	Server
FriendRequest	Ask if a friend is registered	Server
ShareListRequest	Share a list with a friend	Server
UnSharelistRequest	Unshare a list with a friend	Server
CreateSharedListRequest	Ask the client to create a shared list (only class which is generated by the server)	Client / Server
ItemRequest	Changes on Items	Client
RenameListRequest	Renaming of shared lists (not yet implemented)	Client

The Client class Requests are subtypes of ListChangeRequest which is a subtype of Request.
A ListChangeRequest contains a unique list ID of the list you want to change.

3 Communication Procedure

3.1 Adding requests

Whenever you make a change of something that should be shared or needs shared information the application adds a request of required type to the RequestQueue.

3.2 Sending requests

Whenever you want to you can call the synchronise method of the syncmanager and all your queued requests get sent to the server.

This whole procedure runs on a seperate thread to not block the UI during the synchronisation.

3.3 Receiving answers

On every sending of requests a receiving of answers follows. The answers get passed to the AnswerHandler, which changes the neccessary things in the core or the UI.

3.4 Updating the userinterface

After the synchronisation thread stopped the userinterface gets updated.

4 Example (Friendslist)

For each of my friends I want to know if he is registered on the server:

```
for(Friend f: friendsManager.getFriendsList()) {
    FriendRequest fr = new FriendRequest(f);
    syncManager.addRequest(fr);
}
syncManager.synchronise(this);
```

For each friend in your friendslist a new Request gets generated and added to the syncmanager. After that you call the method synchronise() which sends all those requests to the server and awaits an answer. The answers get passed to the AnswerHandler which checks if they were successful (the successful flag gets set by the server):

```
for (Request r: requests) {
    if(r.wasSuccessful()) {
        setConsequences(r);
```

If they were successful, consequences are set:

```
case Request.FRIEND_REQUEST:
    long friendId = ((FriendRequest)request).getFriendId();
    friendsManager.setFriendHasApp(friendId);
    return;
```

The last method which is always called after the synchronisation thread stopped is:

```
public void updateUI() {
    context.refresh();
}
```

It updates the userinterface. In our example this would be the friendslist.

For every other type of request it looks quite the same.