PalPointer: Utvecklarmanual

PalPointer en enkel applikation som hjälper användaren att hitta sina vänner. En pil som visar riktning samt distans kommer upp på skärmen så användaren vet vart och hur långt han eller hon ska gå för att hitta sin vän.

För vidare utveckling av PalPointer krävs ett Java 6 SE development environment, Android SDK och en, eller allra helst två androidenheter, att testa applikationen på. Ladda ner appen via:

```
git clone git://github.com/DAT255-GLPP/AppPalPointer.git
```

Android SDK targets:

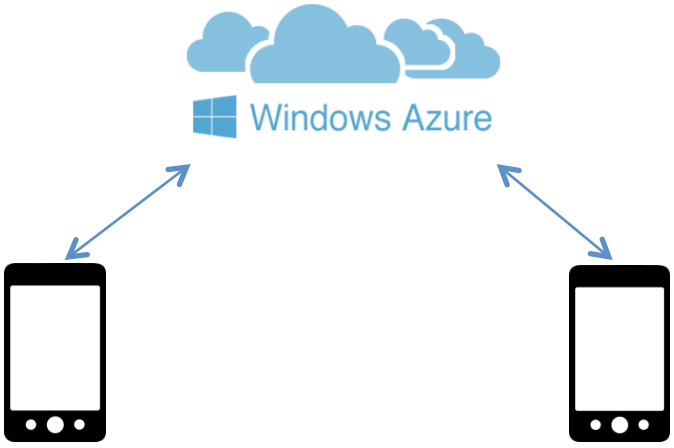
- Minimum SDK: 8
- Target SDK: 18

Design och arkitektur

PalPointer är uppbyggd med utgångspunkt i GRASP (General Responsibility Assignment Software Patterns), vilket beskriver olika grundläggande principer och riktlinjer för ansvarsfördelning bland klasser och objekt. GRASP består av nio olika mönster. Ett av dessa mönster är information expert, vilket beskriver var ansvaret för olika uppgifter ska ligga. Enligt information expert ska den klass som har tillräckligt med information vara ansvarig för uppgiften. Detta har vi försökt utgå hela tiden när vi skapat våra metoder. Något annat vi har försökt att uppnå är low coupling och high cohesion. Bland annat har vi valt att använda mer avancerade kopplingar med lyssnare istället för OnClick-metoder i xml-filerna, så att kopplingarna minskar.

Central databas via Microsoft Azure

För att hantera kommunikationen mellan de mobila enheterna har en central databas kallad Windows Azure använts. När en användare trycker på knappen ”Upload my position” laddas personens koordinater kontinuerligt upp till databasen tills användaren aktivt avbryter detta. En person som söker efter en vän laddar samtidigt ner dessa koordinater från databasen med jämna mellanrum. Det går även att både ladda upp sina egna koordinater till databasen samtidigt som nedladdning av vännens position sker. Både uppladdning och nedladdning sker med hjälp av trådar som beskrivs mer utförligt i ett separat avsnitt.



För att använda Azure i appen finns ett objekt av typen *MobileServiceClient* som kopplar appen till vår skapade sida på Azure med hjälp av en URL-adress och en nyckelsträng. Databastabellen som används är av en typ som av Microsoft valt att kalla *MobileServiceTable* och består i vårt fall av objekt av klassen *UserInformation*. En entry i databasen består i sin tur av åtta kolumner. Fyra av dessa är fördefinerade av Microsoft och fyller i dagsläget ingen egentlig funktion i appen. De kolumner som har lagts till av oss är *userid, phonenumber, latitude* och *longitude.* Dessa fyra samt kolumnen *id* finns definierade som variabler klassen UserInformation. Userid är ett unikt id genererat av Facebook vid inloggning till appen. Med hjälp av userid kan varje användare av appen särskiljas. Inloggningen hanteras i metoden *authenticate().* Om det är användarens första inloggning tvingas denne ange ett phonenumber som kopplas till det aktuella userid:t. När användaren laddar upp sin position uppdateras värdena i kolumnerna för latitude och longitude. En entry ser ut på följande vis.

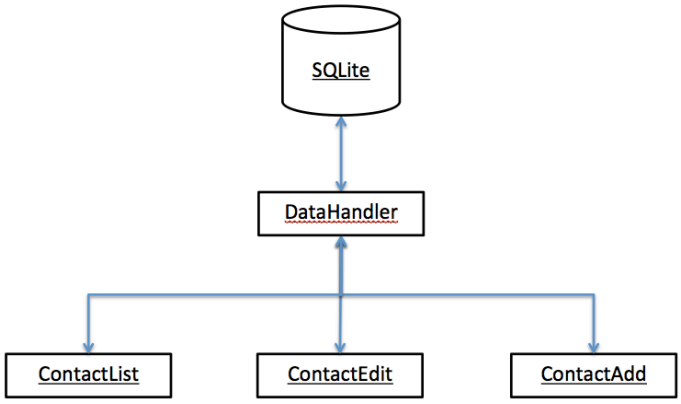
id	userid	phonenumber	latitude	longitude
031F6F7A-703B-4589-9C19...	Facebook:1452148335032559	7966467815	-1000	-1000

__createdAt	__updatedAt	__version
2014-06-02T09:55:46.608+0...	2014-06-02T10:24:05.155+0...	AAAAAAABcrY=

Lokal databas genom SQLite

För att användare av PalPointer inte ska behöva fylla i ny information varje gång appen startas ges möjlighet att skapa och spara kontakter inuti själva appen. Detta görs med hjälp av en lokal databas, nämligen SQLite, som sparas på telefonen.

I syfte att undvika att ha metoder överallt som kommunicerar direkt med denna lokala databas har klassen *DataHandler* skapats. *DataHandler* fungerar som ett gränssnitt mellan SQLite och resten av appen. Detta betyder att övriga klasser kan kommunicera med *DataHandler* som i sin tur kommunicerar med den lokala databasen, vilket illustreras i figuren till höger.



Av denna anledning innehåller *DataHandler* flera metoder som kan anropas av övriga klasser. Metoderna i *DataHandler* som anropas är av två typer. Dels metoder för att lägga till, ta bort och uppdatera data och dels en metod som läser och returnerar data från databasen. Metoden för att lägga till data används av klassen *ContactAdd*, metoderna för att ta bort och uppdatera data används av *ContactEdit* och metoden för att returnera data används av *ContactList*.

För att *DataHandler* ska ha åtkomst till SQLite och ha någonting att köra sina metoder på skapas i denna klass en instans av *SQLiteDatabase*. Vidare innehåller *DataHandler* en inre klass vid namn *DataBaseHelper*, som utökar *SQLiteOpenHelper*. Denna används för att underlätta och öka robustheten vid eventuella uppgraderingar.

Kontinuerlig uppladdning

För att PalPointer ska kunna fungera på avsett sätt krävs det att applikationen klarar av att utföra flera saker samtidigt, i bakgrunden. Detta innefattar upp- och nedladdning av gps-position, den egna positionen laddas upp kontinuerligt medan din väns position laddas ner kontinuerligt. Denna information används för att beräkna distans och bäring (riktning). Vi har valt att hantera detta genom att låta flera trådar köras parallellt.

För att åstadkomma detta har vi skapat klassen *UpdatingThreads*, vilken är en subklass till klassen *Thread*. Förutom en konstruktor, och den ”obligatoriska” metoden run(), innefattar denna klass framför allt två olika metoder, ”downloadPalsCoordinates()” och ”uploadOwnCoordinates()”. Beroende på vilken typ av tråd som skapas kommer antingen nedladdning av en väns position eller uppladdning av den egna positionen starta då en instans av *UpdatingThreads* skapas och metoden run() körs.

Båda dessa trådar körs kontinuerligt, men ”sover” periodvis för att invänta respons från databasen Microsoft Azure (detta för att inte flera förfrågningar till databasen ska hamna på kö). Trådarna körs kontinuerligt tills dess att de manuellt stängs av (då det gäller nedladdning av en väns position finns denna funktion för närvarande inte i applikationen beroende på att vi anser att detta ej är önskvärt, däremot är koden förberedd för att åstadkomma detta om så önskas). Den manuella avstängningen (i detta fall avstängningen

av att ladda upp de egna koordinaterna) sker genom att instansvariabeln ”boolean executeWhileLoop” ändras från true till false, i detta fall genom en knapptryckning på huvudskärmen. Trådarna körs med andra ord kontinuerligt genom att uppdateringen innesluts av en while-loop med värdet true, tills dess att detta ändras.

UML

Som UML-schemat (UML.png) visar är appen huvudsakligen uppbyggd av två klassgrupper. I den ena hanteras allt som involverar kontaktfunktionen. Här finns samtliga aktiviteter som har med kontakthantering att göra, det vill säga *ContactList*, *ContactEdit* och *ContactAdd*. Dessa tre är sammankopplade med *DataHandler* som är länken till databasen som innehåller kontaktlistan.

Den andra klassgruppen har i princip hand om resten av appens funktionalitet. Ryggraden i appen är aktiviteten *ToDoActivity*. Från denna kan flera andra aktiviteter anropas genom knapptryck och den är även kopplade till klassen *UpdatingThreads* som sköter upp- och nedladdning av koordinater. Klasserna *UserInformation* och *Authenticate* är viktiga för hanteringen av Azure.