

Valfri Applikation - PicChat



Khalil Younis

Khalilkanade_5@hotmail.com

Utveckling av mobila applikationer

VT-21

Innehållsförteckning

1. Introduktion.....	3
2. Beskrivning av kamera och säkerhet.....	4
<i>2.1. Kamera.....</i>	<i>4</i>
<i>2.2. Dataanvändning</i>	<i>4</i>
2.2.1. Authentication	4
2.2.2. Realtime Database	4
2.2.3. Storage	5
<i>2.3. Sammanfattning av dataanvändning</i>	<i>5</i>
3. Manual för användare	6
4. Interaktion av beståndsdelar	10
5. Problem	12
6. Källförteckning.....	15

1. Introduktion

Syftet med projektet är att skapa en gratis respektive annonsfri *Snapchat* liknande app, där användaren kan lägga till/ta bort vänner, skicka samt ta emot bilder från vänner. Det är dock inte möjligt att lägga till en bildtext till bilden man vill skicka iväg. Det sägs att en bild säger mer än tusenord, därför anses bildtexten vara onödig. Uttryck dig själv med meningsfulla, roliga samt kreativa bilder och skicka iväg meddelandet enkelt och smidigt med *PicChat*.

Bildmeddelanden som skickas är sparade i en så kallad Realtime Database skapad med hjälp av Firebase, som endast mottagaren har åtkomst till. Bildmeddelanden tas bort från databasen så fort mottagaren har öppnat det. Appen är avsedd för alla, även barn dock i förälders sällskap. Betyget som jag siktar på med min lösning är VG.

2. Beskrivning av kamera och säkerhet

2.1. Kamera

Det finns flera olika sätt att komma åt bilder från kameran. Med tanke på att vi är endast intresserade av att ta en enkel bild så kan man göra detta genom att starta en redan befintlig aktivitet som hanterar detta. Detta görs genom att starta intentet *MediaStore.ACTION_IMAGE_CAPTURE*. Ingen permission behövs för att starta denna intent, för att kameraappen redan har tillstånd att ta bilder.

PicChat använder sig av tre olika typer av tjänster som *FireBase* erbjuder, *Realtime Database*, *Storage* samt *Authentication*. Data som sparas av *PicChat*-användare är följande: *Email*, *Password*, *Full Name*, *UserName* samt bildmeddelanden.

2.2. Dataanvändning

2.2.1. Authentication

Autentisering sker med hjälp av *FireBase* tjänsten *Authentication*, dvs användarens lösenord sparas inte i *Realtime Database*, lösenordet är varken synlig för andra användare eller för apputvecklaren. Ett konto registreras med *Email*-adress samt lösenord, sedan skapas ett unikt authentication-ID som binds med kontot. Genom användningen av *Authentication* tjänsten kan *Realtime Database* identifiera användaren med hjälp av det unika ID:et.

2.2.2. Realtime Database

Realtime Database innehåller följande grenar: *Users*, *userNames*, *Requests* samt *Sent*. När en ny användare ska skapa ett konto måste appen gå igenom alla *userNames* i databasen, för att säkerställa att användarnamnet som ska skapas är unikt, dvs att användarnamnet inte är upptaget. På grund av denna anledning är *userNames* grenen tillgänglig att läsas för alla, detta ger dock inte åtkomst till annan info än själva användarnamnet. Som sagt skapas det ett unikt ID i *Authentication* och användarnamnet läggs till i grenen *userNames* när ett nytt konto registreras. Därefter skapas en child-element i grenen *Users*, där det unika ID:et, användarnamnet samt fullständiga namn sparas. *Users*-grenen har begränsad åtkomst, dvs. endast användare med samma ID som kontot är kopplad till kan läsa samt ändra innehållet.

När en användare ska lägga till en vän kan hen endast läsa användarnamnet från databasen. När en vänförfrågan skickas sparas det i grenen *Requests*, där ID:et för sändaren samt ID:et för mottagaren är en del av förfrågan. *Requests* grenen har annorlunda åtkomstregler jämfört med de andra grenarna. Det är såhär att alla användare kan skicka vänförfrågningar till vem som helst i systemet, därför kan alla registrerade användare skriva i denna gren oavsett ID. Däremot kan endast mottagaren läsa förfrågningar registrerade på sitt unika ID. När en förfrågan accepteras kommer en child-element att skrivas i grenen *Users/Uid/friendsList/*, för att vänner ska kunna visas i vänlistan samt för att man ska kunna skicka bildmeddelanden till dessa. En utmaning jag stötte på var att när en förfrågan skulle accepteras skulle både sändaren av förfrågan samt mottagaren få varandras uppgifter registrerade i varsitt friendsList. Detta innebär att när en förfrågan accepteras ska mottagaren kunna skriva i sändarens friendsList, vilket i sin tur betyder att användare ska ha åtkomst till andras data. Detta löstes med hjälp av begränsning till vad användare kan skriva i andras data i databasen. Alltså när förfrågan accepteras kommer mottagaren att endast kunna skriva sina egna uppgifter i sändarens friendsList. Detta kan endast ske om sändaren har skickat en vänförfrågan, vilket gör det omöjligt för vem som helst att kunna lägga till data i andra användares ”gren” utan en förfrågan.

Vid överföring av meddelanden sparas sändarens-, mottagarens- respektive bildmeddelandets-ID i *Sent* grenen. Begränsningen i denna gren är att enbart mottagaren kan läsa vad som skickats till hen samt att endast vänner till mottagaren kan skicka iväg meddelanden.

2.2.3. Storage

I grenen *Sent* i Realtime Database sparas endast referenser till meddelanden som skickats. Storage-tjänsten används för att spara bilderna (.JPEG) och använder sig av samma referenser som i Realtime Database. När mottagaren öppnar sitt meddelande kommer den att tas bort både från Realtime Database samt Storage.

2.3. Sammanfattning av dataanvändning

Data som användaren matar in är användarnamn, email-adress, lösenord samt fullständiga namn. Email-adressen är till för att användare ska kunna verifiera sina konton samt återställa sitt lösenord. Email-adressen är inte synlig för någon förutom utvecklaren, däremot är lösenordet osynligt. När andra användare ska lägga till en som vän kan de endast se

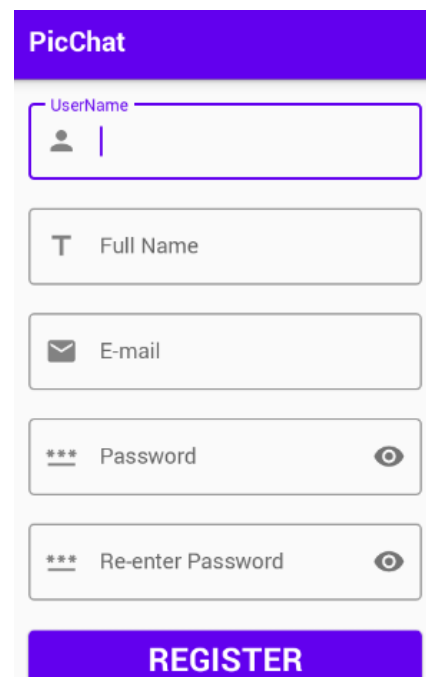
användarnamnet, ifall de blir accepterade får de åtkomst till ens fullständiga namn. Bilder som skickas sparas i *Storage* som en .JPEG-fil samt i *Realtime Database* i form av referens. När mottagaren har öppnat meddelandet försvinner meddelandet från systemet.

3. Manual för användare

Efter att användaren har registrerat samt verifierat sitt konto genom email, får användaren möjligheten att logga in. Figurerna 1 respektive 2 visar de olika aktiviteterna för registrering samt inloggning.




Figur 1: Startsidan för appen



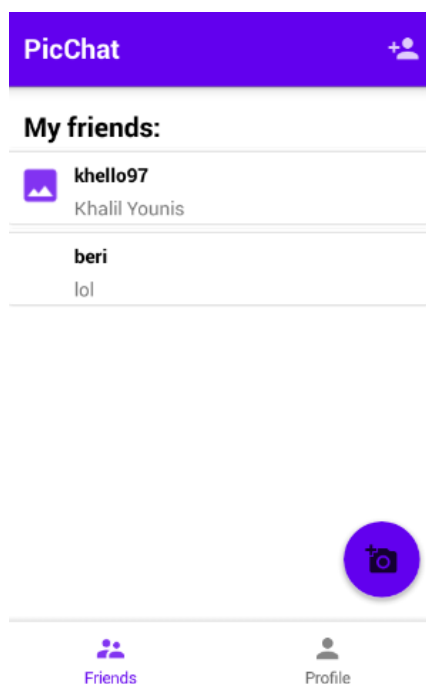
Figur 2: Registreringsvyn

Ifall användaren har glömt bort sitt lösenord kan man enkelt återställa det genom att fylla in sin email-adress. Kort därefter får användaren ett mejl på sitt konto med en länk där användaren erbjuds återställa sitt lösenord. Se figur 3.

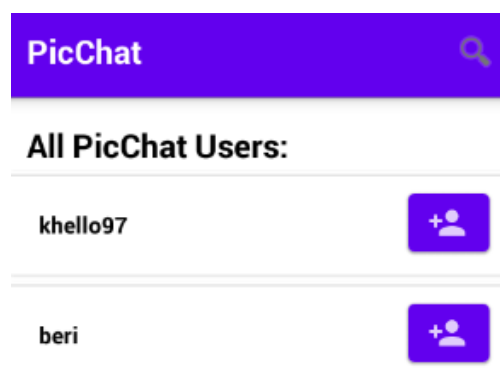


Figur 3: Vyn för återställning av lösenord

Figur 4 visar vyn för användaren efter ett lyckat inloggningsförsök. Som figuren visar finns det en så kallad *bottomNavigationBar* där man kan pendla genom två olika fragment: *Friends* respektive *Profile*. Man kan även lägga till vänner genom en item som befinner sig ovan i *actionBar*, vilket tar användaren till *addFriends* aktiviteten (figur 5). Man kan antingen scrolla i listan eller söka efter en specifik användare genom sökfunktionen i *actionBar*. Lägga märke till att endast användarnamnet är synligt då man ska lägga till vänner. Ifall vänförfrågan accepteras kommer den nya vännen att läggas till i ”My friends”-listan (figur 4), där det fullständiga namnet är inkluderat. I figur 4 ser man en lila symbol bredvid en vän i listan, detta indikerar att man har fått ett bildmeddelande från den personen. Man öppnar meddelandet genom att enkelt klicka på användaren i listan. Det kommer föra oss vidare till en ny aktivitet där bilden visas i fullskärm. Genom att klicka på antingen ”back”-knappen eller på själva bilden så förs man tillbaka till *Friends*-fragmentet. Observera att bilden är fel roterat om man använder sig av en emulator. Denna funktion är testad på flera olika fysiska enheter där bilden visas med rätt rotation.

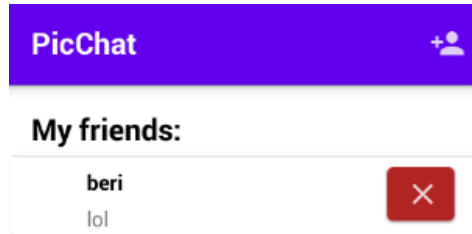


Figur 4: *Friends*-fragment



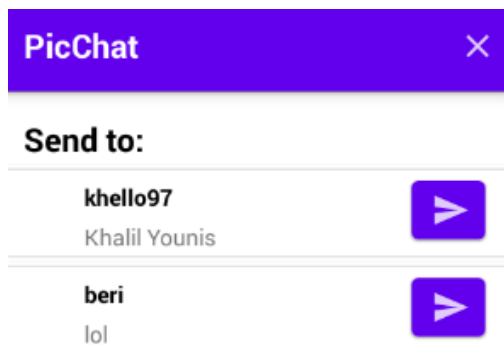
Figur 5: *AddFriends*-vyn

Ifall man vill ta bort en befintlig vän så gäller det att ”lång-trycka” på en användare i vänlistan tills en röd knapp dyker upp. Klickar man på den så tas användaren bort från vänlistan samtidigt som du tas bort från deras vänlista. Se figur 6.



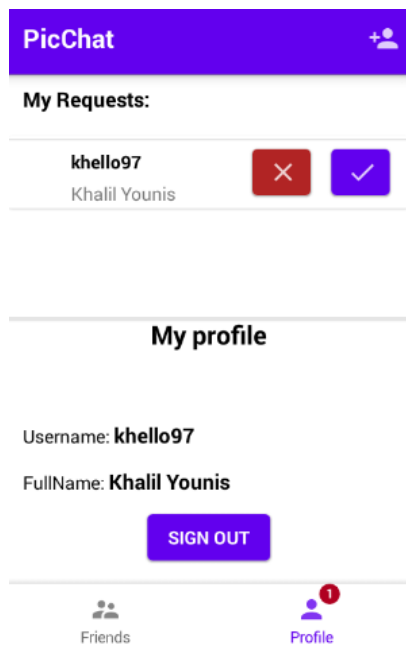
Figur 6: Den röda knappen som dyker upp vid borttagning av vän

FloatingActionButton (kameraknappen i figur 4) tar användaren till en kamera intent där användaren får ta en bild. Sedan bes användaren att välja vänner som bilden ska skickas till. Se figur 7.



Figur 7: Lista med vänner som användaren kan skicka bild till

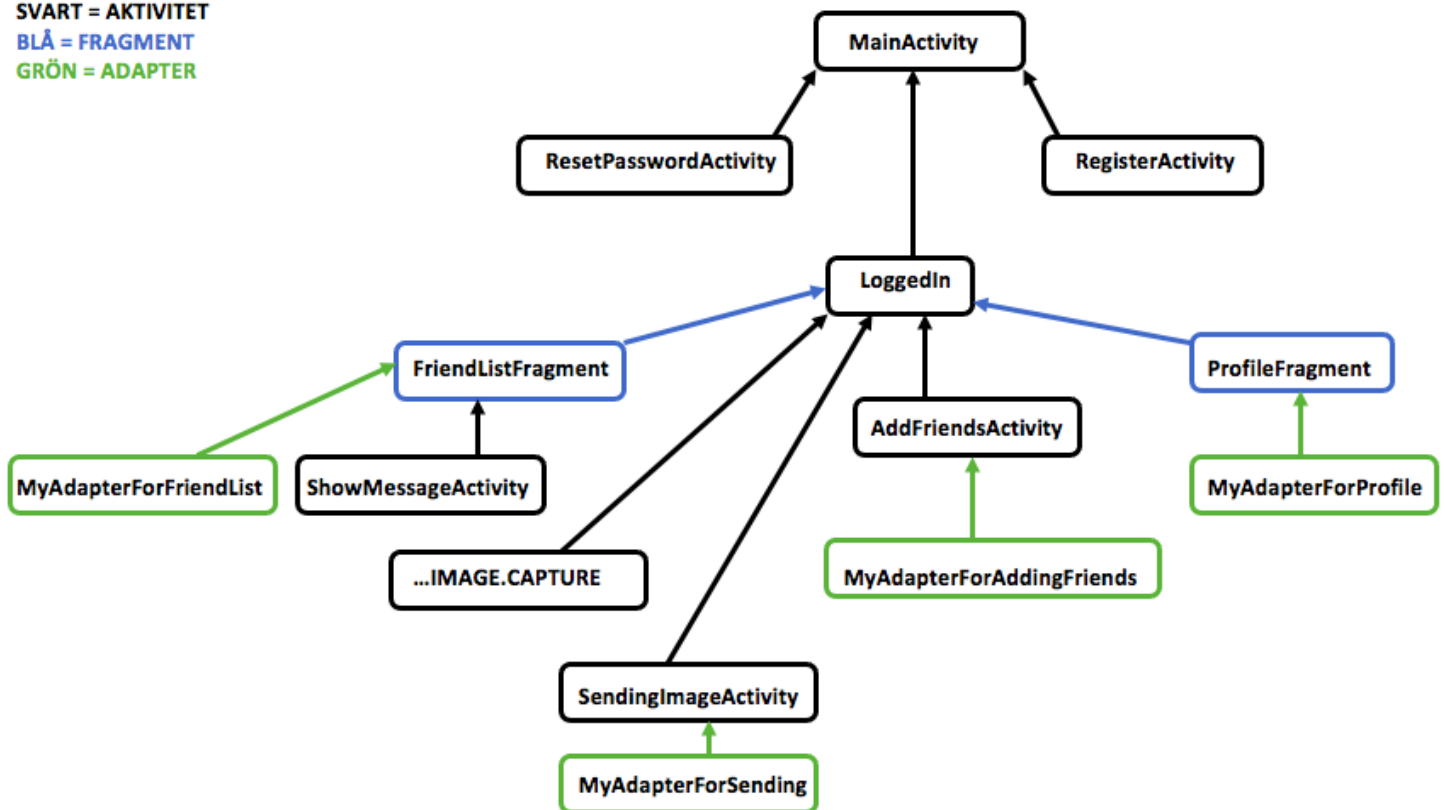
Profile-fragmentet visas i figur 8. Om användaren har pågående vänförfrågningar så presenteras dessa under *My Requests*, användaren kan antingen acceptera eller neka förfrågan. Ifall man har pågående vänförfrågningar så visas ett så kallat *badge*-nummer med antalet förfrågningar, längst ner på *Profile*-ikonen. Under *My Requests*-listan hittar användaren info om sitt eget konto såsom, fullständiga namn samt användarnamn. Sist men inte minst befinner sig logga ut knappen i *Profile*-fragmentet, där användaren loggas ut från sitt konto och skickas tillbaka till inloggningsvyn (figur 1).



Figur 8: *Profile*-fragment

4. Interaktion av beståndsdelar

SVART = AKTIVITET
BLÅ = FRAGMENT
GRÖN = ADAPTER



Figur 9: UML-Diagram som visar interaktionen av beståndsdelarna

Varje aktivitet eller fragment som innehåller en lista, använder sig utav en specifik adapter. Detta pga. att en lista består av en så kallad *recyclerView* som innehåller alla element som visas för användaren. En *recyclerView* innehåller flera *viewHolders*, där varje *viewHolder* är ett element i listan som innehåller alla komponenter rörande just det elementet. Data som visas i dessa listor som t.ex. användarnamn, fullständiga namn mm. befinner sig i olika referenser i databasen. Det som binder data från databasen med listorna är en adapter. Med tanke på att data ska hämtas från flera olika referenser från databasen för varje lista, bestämdes det att skapa en adapter till varje lista. Egentligen hade man kunnat använda en adapter till flera listor, men resultatet skulle vara en ostrukturerad kod. Detta underlättar för både utvecklaren samt för andra att hänga med kodmässigt.

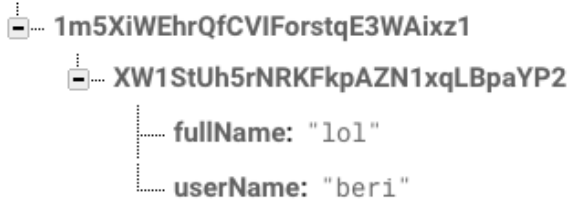
Ett exempel på skillnad mellan listor, komponentmässigt, är att *My Request*-listan (figur 8) som befinner sig i *ProfileFragment* innehåller två olika knappar, acceptera respektive neka, medan *My friends*-listan (figur 4) som befinner sig i *FriendListFragment* innehåller *TextViews* som framställer användarnamn respektive fullständiga namn. *My friends*-listan får sin data från referensen `"/Users/currentUserId/friendsList"`, medan *My Request* får sin data från referensen `"/Requests/currentUserId"`. Listorna använder sig utav olika referenser från databasen, därför anses det vara mycket smidigare att använda en adapter till varje lista.

Observera att när användaren har tagit en bild i kameraaktiviteten (`...IMAGE.CAPTURE`) så förs användaren först tillbaka till *LoggedIn*-aktiviteten där bilden sparas i en *byteArray*, därefter skickas *byteArray*en men en *intent* som startar *SendingImageActivity*. Från användarens perspektiv ser det ut som att *SendingImageActivity* startas direkt efter kameraaktiviteten, men i UML-diagrammet ovan ser man hur *LoggedIn*, `...IMAGE.CAPTURE` samt *SendingImageActivity* interagerar med varandra.

5. Problem

Det uppstod massor av olika typer av problem under laborationens gång. En av de största utmaningarna var att bestämma databasreglerna, för att göra det så säkert som möjligt för användare. Innan jag går igenom reglerna för databasen så måste vi först ta en titt på hur databasen är uppbyggd med hjälp av nedanstående figurer.

Requests



Figur 10: Request-grenen

Users



Figur 11: Users-grenen

I figur 10 kan vi se en vänförfrågan som har skickats. Första ID numret tillhör mottagaren till vänförfrågan, det andra ID:et tillhör sändaren av förfrågan. Därefter kommer användarnamnet samt fullständiga namnet till sändaren av förfrågan. Som sagt i sektion 2.2.2. *Realtime Database*, så kan vilken användare som helst skicka vänförfrågningar till alla, därför kan alla skriva i Request-grenen. Däremot kan endast användare med samma ID som mottagaren till förfrågan läsa dessa requests.

I figur 11 kan man se hur Users-grenen i databasen är uppbyggd. Först kommer ID:et till användaren, sedan friendsList där användarens vänner befinner sig. Sist står det fullName samt userName som tillhör användaren. Endast användare med samma ID har åtkomst till sin egna data, både när det kommer read- respektive write rättigheter.

Problemet som uppstår då är att ifall en användare ska acceptera en vänförfrågan så måste denna användare kunna skriva i både sin egen respektive sändarens `friendsList`, så att båda har varandra som vänner. Detta bryter mot Users-grenens regler. Det finns en tjänst av FireBase vid namnet *Functions*, där man kan skriva egna funktioner i databasen som triggas vid specifika event som exempelvis att acceptera en vänförfrågan. Då skulle databasen kunna skriva i andra användarens `friendsList` utan att någon regel bryts. Men tyvärr så är denna tjänst inte gratis att använda, problemet löstes därför på ett lite annorlunda sätt.

```
"Users":{  
  
  "$user_id":{  
    ".read": "$user_id == auth.uid",  
    ".write": "$user_id == auth.uid",  
  
    "friendsList":{  
      "$friend_id":{  
        ".write": "$friend_id == auth.uid"  
      }  
    }  
  }  
},
```

Figur 12: Regler för Users-grenen

Som figur 12 visar har användare med authentication ID som matchar *user_id* rättigheten att läsa samt skriva, med en liten twist. *FriendsList* ger rättigheten för andra användare att skriva, endast om de lägger till sig själva. Regeln i figuren visar att *friend_id* måste matcha authentication ID till användaren. Detta är dock endast möjligt om användaren som ska skriva i andras *friendsList* har fått en vänförfrågan.

Ett annat problem som jag stötte på var användningen av funktioner som FireBase biblioteket erbjöd. Det tog lång tid tills jag förstod hur *ValueEventListener* mm. fungerade. Det är nämligen så att när man lägger till en *ValueEventListener* på en referens, så körs det en gång. Sedan körs det igen för varje gång data skrivs i denna referens, vilket ibland kunde bli fel. Detta löstes enkelt genom att använda sig av *boolean* variabler, vilket begränsade *ValueEventListener* från att köras mer än nödvändigt.

Vid utloggning av kontot överförs användaren tillbaka till *MainActivity*. Problemet var att enheten aldrig stängde av gamla aktiviteter som hade körts tidigare. Det påverkade enhetens prestanda markant. Ett annat problem var att efter att man hade loggat ut så kunde användaren klicka på back-knappen, appen skulle då vilja gå tillbaka till LoggedIn-aktiviteten, vilket kraschade appen. Detta löstes genom att rensa activity-stacken efter utloggning. Ifall back-knappen trycks så går enheten tillbaka till *Home* (utanför appen).

Istället för att ha många liknande layouter till olika *viewHolders*, så har en layout, *list_item_friend.xml*, återanvänts av flera olika *viewHolders*. Komponenterna som inte passade för en specifik *viewHolder* sattes *visibility* till *gone*. På så sätt återanvändes samma layout till olika *viewHolders*. Med tanke på att min lösning använder sig av en databas så har nästan inga *states* behövts sparas. Detta pga. att nästan alla variabler får sina värden direkt från databasen, därför är alla variabler alltid uppdaterade.

Nackdelen med lösningen är att kvalitén på bilden blir dålig vid mottagning. Kvalitén försämras när en användare ska ladda ner/hämta bilden från *Storage*. Beviset på detta är att bilden har minst lika bra kvalité när den befinner sig i *Storage*. Den stora biten av min lösning handlade om databasens regler, interaktionen mellan beståndsdelar mm. Det man skulle kunna göra för att förbättra lösningen ytterligare är att förbättra kvalitén på bilden, förbättra *addFriends* algoritmen, så att det blir omöjligt att skicka iväg vänförfrågan till redan befintliga vänner. Man skulle även kunna utöka funktionerna och göra det möjligt att skicka små filmklipp till varandra.

6. Källförteckning

Inga källor användes. Alla figurer och bilder är tagna från emulatore i Android Studio. UML-diagrammet har jag skapat själv med hjälp av microsoft word. Även bilderna som visar hur databasen är uppbyggd har jag själv skapat med hjälp av FireBase.