0.1 Implementering af persistens

Datapersistering og datahentning er vigtig komponent i dette system. Implementering af persistens vil derfor blive beskrevet meget nøje, og herunder delt op i tre dele; Implementering af persistens i mobilapplikation, Implementering af persistens i simulator og Implementering af persistens i online værktøjet. Hver del vil ikke have en beskrivelse af den fulde implementering, men blot repræsenteret af væsentlige dele. For fuld implementering af persistens henvises der til bilags CDen, i den respektive komponent under mappen Kode.

0.1.1 Implementering af persistens i mobilapplikationen

Persistering i denne komponent falder i to underpunkter. Dette er fordi, denne komponent er den eneste, som har kontakt til to databaser; Den distribuerede MySQL database samt den lokale SQLite database. Disse to vil blive beskrevet i seperate afsnit.

Tilgang til MySQL databasen

Mobilapplikationen har aldrig direkte tilgang til den distribuerede database. Tilgangen sker i afsnittet *8.2.2: Komponent 2: Mobil service*.

Applikation kommunikerer med databasen igennem en service, og altid kun som en læsning. Dette gør det muligt at tilgå databasen fra flere enheder, da en database læsning er trådsikker. Grunden til at der bliver gjort brug af en service er, at database tilgangen skal kunne gemmes væk fra brugeren, således en person ikke kan få fuld tilgang til databasen igennem sin mobil. Desuden vil mobil applikationen nemt kunne skiftes ud, uden at skulle tænke på tilgangen til databasen.

Selve kommunikationen med servicen sker igennem en SoapProvider. SOAP bruges som en transportmetode til XML beskeder. Når mobilen tilgår servicen opretter den en SOAP-envelope, der indeholder information om, hvilken metoden der skal kaldes, under hvilket namespace metoden ligger, samt eventuelle parametre metoden. På kodeudsnit 1 kan en generisk oprettelse og transmitering af en SoapEnvelope ses.

Kodeudsnit 1: Generisk SoapEnvelope.

```
1 SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);
2 request.addProperty(PARAMETER_NAME, PARAMETER_VALUE);
3 SoapSerializationEnvelope envelope = new ↩
    SoapSerializationEnvelope(SoapEnvelope.VER11);
4 envelope.dotNet = true;
5 envelope.setOutputSoapObject(request);
6 HttpTransportSE androidHttpTransport = new HttpTransportSE(↩
    URL_OF_SERVICE);
7 androidHttpTransport.call(NAMESPACE+METHOD_NAME, envelope);
8 SoapObject response = (SoapObject)envelope.getResponse();
```

Requestet oprettes som et SoapObject, hvor metodenavnet, samt det namespace metoden ligger i, gives med. Disse to parametre er strings. Til metodekaldet kan der tilføjes parametre ved "addProperty"metode, som tager imod et parameter navn og en parameter værdi, begge to strings. Envelopen bliver oprettet og et versionsnummer bliver givet med, der definerer hvilken version af protokollen der skal tages i brug. I dette system bliver der udelukkende gjort brug af version 1.1. dotNet flaget er sat til true, da servicen er skabt i ASP.NET. Request-objektet sættes i envelopen, og kommunikerer med servicen over HTTP. Efter den relevante metode er færdigjort på servicen bliver returværdien sat i envelopen, og et SoapObject indeholdende de returnerede værdier kan hentes ved et kald til "getResponse"metoden på envelopen.

Et SoapObject er reelt set et XML-træ, som kan itereres igennem. Et eksempel på et sådan XML-struktur kan ses i afsnitet *8.2.2: Komponent 2: Mobil service*

Et fuldt eksempel på et kald til servicen kan ses på kodeudsnit 2. Denne funktion bruges til at hente samtlige busser med et givent busnummer, og returnere dem som en ArrayList.

Kodeudsnit 2: GetBusPos. Returnerer alle bussers position på en given rute.

```
1
2 final String NAMESPACE = "http://TrackABus.dk/Webservice/";
3 final String URL = "http://trackabus.dk/AndroidToMySQLWebService.↩
    asmx";
4 ...
5 public ArrayList<LatLng> GetBusPos(String BusNumber)
6 {
7     ArrayList<LatLng> BusPoint = new ArrayList<LatLng>();
8     try
9     {
```

```
10          SoapObject request = new SoapObject(NAMESPACE, "GetbusPos↩
               ");
11          request.addProperty("busNumber", BusNumber);
12          SoapSerializationEnvelope envelope = new ↩
               SoapSerializationEnvelope(SoapEnvelope.VER11);
13          envelope.dotNet = true;
14          envelope.setOutputSoapObject(request);
15        HttpTransportSE androidHttpTransport = new HttpTransportSE(↩
             URL);
16        androidHttpTransport.call(NAMESPACE+"GetbusPos", envelope);
17        SoapObject response = (SoapObject)envelope.getResponse();
18
19        for(int i = 0; i<response.getPropertyCount(); i++)
20        {
21          double a = Double.parseDouble(((SoapObject)response.↩
               getProperty(i)).getProperty(0).toString().replace(",",↩
                "."));
22          double b = Double.parseDouble(((SoapObject)response.↩
               getProperty(i)).getProperty(1).toString().replace(",",↩
                "."));
23          BusPoint.add(new LatLng(a, b));
24        }
25      }
26      catch(Exception e)
27      {
28        return null;
29      }
30      return BusPoint;
31  }
```

Metoden på servicen returnerer en liste, indeholdende typen "Point", som er en custom datatype lavet i servicen. Denne har to attributer, Latitude og Longitude, som begge er strings. "getPropertyCount"funktionen returner længden af denne liste, og bruges til at iterere igennem den.

Det første kald af "getProperty"på responset, returnerer "Point"datatypen. Denne property castes til et nyt SoapObjekt, hvor "getPropety"kaldes igen. Rækkefølgen af propeties i et SoapObject, defineres af rækkefølgen de bliver oprettet i, i datatypen. I "Point"kommer latitude først og longitude kommer bagefter. "GetProperty(0)"på et "Point"SoapObjekt vil derfor returnere Latitude og "GetProperty(1)"vil returnere Longitude. Begge bliver castet til en string, og floatingpoint sættes til et dot frem for et komma. Dette gøres da ASP.NET tager et floatingpoint som værende komma.

Da applikationen er lavet til Android bruges biblioteket ksoap2[1], som er specifik for Android. I dette bibliotek ligger alle funktioner, der er nødvendige for at bruge af SOAP.

Tilgang til SQLite databasen

Når en busrute favoriseres gemmes alt data om denne i en lokal SQLite database. Dette gøres for at spare dataforbrug, hvis en rute tages i brug ofte. Samtidig muligøres det også, at brugeren kan indlæse en rute med stoppestedder, uden at have forbindelse til internettet. Hvis kortet samtidig er cachet (Google Maps cacher indlæste kort), kan kortet også indlæses og indtegnes.

Der er gjort brug af en ContentProvider i denne sammenhæng, som abstraherer data-access laget, så flere applikationer kan tilgå databasen, med den samme protokol, hvis det skulle være nødvendigt.

En ContentProvider tilgås igennem et kald til "getContentResolver", hvorefter der kan kaldes til de implementerede CRUD-operationer. En ContentProvider skal defineres i projektets AndroidManifest, før den kan tilgås. Dette gøres ved at give den et navn, samt en autoritet, som dette tilfælde, er den samme værdi som navnet. Hvis ContentProvideren bruges i en anden applikationen, skal den have en autoritet tilsvarende den, den er blevet oprettet med.

Da ContentProvideren blot er et transportlag mellem brugeren og databasen, er det nødvendigt for den, at kende den egentlige database. Dette er gjort ved at lave en inner class til provideren, som extender SQLiteOpenHelper. Denne klasse indeholder create proceduren, samt muligheden for at kunne tilgå både en læsbar og skrivbar version af databasen. Create proceduren bliver kørt, hvis databasen med det valgte navn ikke eksisterer i forvejen, og bruges til at oprette databasen og tabellerne deri. En SQLite database gør, som default, ikke brug af foreign key constraints. Det er derfor blevet implementeret sådan, at foreign key contraints aktiveres hver gang databasen åbnes.

Hver CRUD-operation modtager et URI, der skal være en kombination af en identifieren "content://", en autoritet (ContentProviderens placering) samt evt. en tabel og en underoperation. Hvis en given CRUD-operation på ContentProvider siden er lavet, sådan at der altid gøre det samme (f.eks. en query der altid returner alt data i den samme tabel), vil identifieren og autoriteten være nok, til at kunne tilgå denne operation. Hvis tilgangen

[1]For mere information, se http://ksoap2.sourceforge.net/

derimod skal være specifik for en given tabel, og evt. underoperation kan en UriMatcher tages i brug. Denne kobler et URI med en given værdi, hvorefter der i operationen kan laves en switch/case der matcher det medsendte URI, og vælger en operation ud fra dette. På 3 ses et eksempel på, hvordan dette er implementeret i systemet. Det skal noteres at dette ikke er komplet implementering, men blot et udsnit. Kommentarer vises ved "!!"

Kodeudsnit 3: GetBusPos. ContentProvider implementering.

```
1 !!ContentProvider class!!
2 public static final String AUTHORITY = "dk.TrackABus.↩
    DataProviders.UserPrefProvider";
3 public static String BUSSTOP_TABLE = "BusStop";
4 private static final int BUSSTOP_CONTEXT = 1;
5 private static final int BUSSTOP_NUM_CONTEXT = 2;
6 ...
7 static {
8   uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
9   uriMatcher.addURI(AUTHORITY, BUSSTOP_TABLE, BUSSTOP_CONTEXT);
10  uriMatcher.addURI(AUTHORITY, BUSSTOP_TABLE+"/#", ↩
      BUSSTOP_NUM_CONTEXT);
11 }
12 ...
13 public Cursor query(Uri uri, String[] projection, String ↩
    selection,
14    String[] selectionArgs, String sortOrder)
15 String Query;
16 SQLiteDatabase db = dbHelper.getReadableDatabase()
17 switch(uriMatcher.match(uri))
18 {
19   case BUSSTOP_CONTEXT:
20     routeID = selection;
21     query = !!Query to get all busstops and their position on a ↩
         route with id being RouteID!!
22     returningCursor =  db.rawQuery(query,null);
23     break;
24   case BUSSTOP_NUM_CONTEXT:
25     stopID = uri.getLastPathSegment();
26     query = !!Query to get a single bustop with id being stopID!!
27     returningCursor =  db.rawQuery(query,null);
28   default
29     return null;
30 }
31 return returningCursor;
```

```
32
33 ...
34 !!BusStop model class!!
35 public static final Uri CONTENT_URI = Uri.parse("content://"
36       + UserPrefProvider.AUTHORITY + "/BusStop");
37
38 ...
39 !!Hentningen af stoppesteder!!
40 getContentResolver().query(UserPrefBusStop.CONTENT_URI, null, ↩
     RouteID, null, null);
41 String StopID = !!Some ID!!
42 String specifikStop = UserPrefBusStop.CONTENT_URI.toString() + "/↩
     "+ StopID;
43 getContentResolver().query(Uri.parse(specifikStop),null,null,null↩
     ,null);
```

Den første del af kodeeksemplet viser oprettelsen af UriMatcheren. Hvis UriMatcheren kender det ID den bliver givet ved i dennes "match"funktion, vil den returne en værdi, der svarer til den, den er blevet givet ved oprettelse. Herefter ses et udsnit af "query"metoden. Hvis URI'et kun indeholder BusStop udover autoriteten, vælges BUSSTOP_CONTEXT, og der hentes alle stoppesteder, som er relevant for den rute der sættes i selection parameteren. Hvis tabellen efterfølges af et nummer i URI'et, vælges BUSSTOP_NUM_CONTEXT, og der hentes kun det stoppested som har det ID sat i URI'et.

Til samtlige tabeller i SQLite databasen er der lavet en model klasse. Disse klasser indeholder kun statiske variabler som definerer den givne tabels kolonner samt den URI, ContentProvideren skal have med, for at kunne tilgå den tabel, modellen definerer.

I sidste del af kodeafsnittet kan det ses, hvordan ContentProvideren tilgås. Det første kald tilgår query funktionen under BUSSTOP_CONTEXT, og henter alle stoppesteder ud for ruten hvor ID'et er "RouteID". Det andet kald tilgår også query funktionen men under BUSSTOP_NUM_CONTEXT, og henter stoppestedet ud hvor ID'et er "StopID".

0.1.2 Implementering af persistens i simulator

Simulatoren implementerer persistens i form af at hente ruter, opdatere hvilken vej en bus kører, samt udregne og persistere ny GPS position for en bus. Samtlige busser kører i deres egen tråd i simulatoren, derfor er det vigtigt at håndtere trådsikkerhed når databasen skal tilgås. DatabaseAccess klassen tager sig af selve databasen tilgangen, og indeholder to

funktioner; En til at skrive til databasen, samt en til at læse. Begge funktioner er statiske, og indeholder en binær semafor, således kun en tråd af gangen kan tilgå databasen. Hvis en tråd allerede er igang med en datahentning eller -skrivning, vil den anden tråd tvinges til at vente, til processen er færdig. Begge funktioner modtager en string, som er den kommando der skal udføres på database. Funktionen der læser fra databasen tager yderligere en liste af strings, som indeholder de kolonner der skal læses fra. Efter fuldent tilgang returneres en liste af strings, med de værdier der er blevet hentet.

Databasen tilgangen bliver håndteret med i librariet MySQL.Data.[2] Da simulatoren er lavet i Visual Studio 2012, og er en WPF-applikation, er der blot gjort brug af NuGet[3] til at hente og tilføje dette bibliotek til programmet. Forbindelsesopsætningen ligger i App.config filen, og hentes ud når der skal bruges en ny forbindelse. På kodeudsnit 4 ses funktionen der læser fra databasen, samt hvordan den tilgås. Kun denne vil vises, da det er den mest interresante. Fuld kode kan findes på bilags CDen under Kode/Simulator.

Kodeudsnit 4: Simulator. Select statement.

```
1 public static bool SelectWait = false;
2 public static List<string> Query(string rawQueryText,List<string>↩
      columns)
3 {
4   while(SelectWait)
5   {
6     Thread.Sleep(10);
7   }
8   SelectWait = true;
9     using(MySqlConnection conn = new MySqlConnection(↩
          ConfigurationManager.ConnectionStrings["TrackABusConn"].↩
          ToString()))
10  {
11    using(MySqlCommand cmd = conn.CreateCommand())
12    {
13      try
14      {
15        List<string> returnList = new List<string>();
16        cmd.CommandText = rawQueryText;
17        conn.Open();
18        MySqlDataReader reader = cmd.ExecuteReader();
```

[2]For mere information, se http://dev.mysql.com/doc/refman/5.6/en/connector-net.html
[3]Et integret værktøj i Visual Studio 2012, til at hente og tilføje biblioteker

```
19          while (reader.Read())
20          {
21            foreach (string c in columns)
22            {
23              returnList.Add(reader[c].ToString());
24            }
25          }
26          reader.Close();
27          conn.Close();
28          SelectWait = false;
29          return returnList;
30        }
31        catch(Exception e)
32        {
33          SelectWait = false;
34          return null;
35        }
36      }
37    }
38 }
39 ...
40 String query = "Select BusRoute.ID from BusRoute";
41 List<string> queryColumns = new List<string>(){"ID"};
42 List<string> returnVal= DatabaseAcces.Query(query, queryColumns);
```

Der ventes i starten af funktionen på, at semaforen frigives. Hvis tråden skal tilgå databasen og en anden tråd allerede er igang, ventes der på, at den låsende tråd gør processen færdig og sætter SelectWait til false.

Når forbindelsen oprettes, gives den en configurations string. Denne string indeholder database navn, server, brugernavn og password, som er alt hvad forbindelsen skal bruge, for at tilgå databasen. Af denne forbindelse laves der en kommando, som indeholder alt den information som skal eksekveres på forbindelsen. Ved kaldet til "ExecuteReader", udføres kommandoen og en reader returneres med de rækker der kunne hentes udfra den givne query. I skrivnings funktionen ville "ExecuteNonQuery", blive kaldt i stedet, da der, i dette tilfælde, ikke skal returneres noget data. Readeren repræsenterer en række i databasen, og når "Read"funktionen kaldes på den, tilgås den næste række. Hvis "Read"returner false, er der ikke flere rækker at læse. Når data skal hentes ud fra readeren, kan der enten vælges at bruge index (kolonne nummeret i rækken), eller kolonnenavn. I dette tilfælde gives samtlige kolonner med som en parameter, og derfor læses der på navn. Til sidst frigøres semaforen og læst data returneres.

I slutningen af kodeudsnittet kan det ses, hvordan denne funktion tilgås. Først laves der en query, som i dette tilfælde henter samtlige busrute IDer. Herefter oprettes der en liste af de kolonner der skal hentes, hvorefter Query funktionen kaldes med begge værdier.

0.1.3 Implementering af persistens i online værktøjet

Online værktøjet består af to dele; Mobil servicen og hjemmesiden. Begge dele er lavet i ASP.NET, og derfor vil database tilgangs proceduren være ens med simulatoren. Servicen står får at lade mobil applikationen tilgå data på MySQL databasen, hvilket også betyder, at funktionerne kun læser data. Ved et kald til servicen vil læst data pakkes ved hjælp af SOAP, som er beskrevet i *9.2.1: Implementering af persistens i mobil applikationen.* Servicen står i midlertid også for at kalde tidsudregnings proceduren på databasen, hvilket er et anderledes kald, end en læsning. På kodeudsnit 5 ses det, hvordan servicen tilgår denne procedure. Det skal noteres at det ikke er den fulde funktion der vises, men blot et udsnit, og derfor kun viser de vigtigste dele. Herved vises der ikke hvordan forbindelsen og kommandoen laves, da oprettelsen er ens med simulatoren.

Kodeudsnit 5: Service. Udsnit af kald til tidsudregnings procedure

```
1 ...
2 cmd.CommandText = "CalcBusToStopTime";
3 cmd.CommandType = System.Data.CommandType.StoredProcedure;
4 ...
5 cmd.Parameters.Add("?stopName", MySqlDbType.VarChar);
6 cmd.Parameters["?stopName"].Value = StopName;
7 cmd.Parameters["?stopName"].Direction = System.Data.↩
    ParameterDirection.Input
8 ...
9 cmd.Parameters.Add(new MySqlParameter("?TimeToStopSecAsc", ↩
    MySqlDbType.Int32));
10 cmd.Parameters["?TimeToStopSecAsc"].Direction = System.Data.↩
    ParameterDirection.Output;
11 ...
12 cmd.ExecuteNonQuery();
13 ...
14 string TimeToStopAsc = cmd.Parameters["?TimeToStopSecAsc"].Value.↩
    ToString();
15 string EndStopAsc = cmd.Parameters["?EndBusStopAsc"].Value.↩
    ToString();
```

I kodeudsnittet kan det ses, hvordan der i kodeudsnittet, i forrige afsnit, blev tilføjet en kommandotext bestående af en MySQL string, nu bliver tilføjet flere værdier til kommandoen. Først og fremmest bliver kommandotypen sat som værende en stored procedure. Herefter kan det ses hvordan både en input og en output parameter bliver sat i kommandoen. Parameterne bliver givet et navn, samt en datatype, hvorefter de gives en værdi hvis de er input parametre. Herefter gives parameteren en retning; Input hvis de er værdier der skal læses i proceduren og output hvis de skal skrives til. Efter proceduren er kørt, vil output parameterne nu kunne læses, med de værdier der er blevet udregnet. I dette tilfæde er der kun vist to parametre, men antallet og deres navne og retning, skal passe overens med den procedure der er lavet på database siden. I afsnittet *9.1.2: Stored Procedures* kan der læses om trådsikkerheden for proceduren.

Da databasetilgangen på hjemmesiden kun er flertrådet når der læses, er systemet trådsikkert. Når der læses vil det altid ske i hovedtråden. Databasen tilgås ligesom servicen og simulatoren ved hjælp af MySql.Data bibilioteket, og tilgås kun i form af simple CRUD-operationer. Der vil derfor ikke vises et kodeeksempel, det dette anses som værende beskrevet i tidligere afsnit. Samtlige funktioner er samlet i DBConnection klassen, som agerer som en Data Acces klasse. Det vil sige, at alle database aktioner tilgås igennem denne klasse.