JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

Faculty of Engineering
and Natural Sciences

Database synchronization between mobile devices and classical relational database management systems

MASTER'S THESIS

submitted in partial fulfillment of the requirements
for the academic degree

Diplom-Ingenieur

in the Master's Program

SOFTWARE ENGINEERING

Submitted by

Gergely Kalapos

At the

Institute for Application Oriented Knowledge Processing

Advisor

a.Univ.-Prof. Dr. Josef Küng

Linz, January, 2015

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Gergely Kalapos
Linz, January 2015

Statutory Declaration

I declare that I have developed and written the enclosed master thesis entirely on my own, and have not used sources or means beyond those that are explicitly declared in the text. All thoughts from others and literal quotations are clearly indicated.

Gergely Kalapos
Linz, January 2015

Abstract

One significant change in the IT industry in the past years was the rapid expansion of mobile devices. As these devices evolve they became part of different information systems and applications running on these devices communicate with other software components as any other software does. Although the performance of these Smartphones and Tablet-PCs also evolved there are some significant differences between classical PCs, Servers and these mobile devices: The latter will never have the same reliable network as a Server in a server room or a desktop PC connecting to a network via a fixed network cable.

Therefore when it comes to database connections the classical “always online” approach does not work in every situation. One solution for the problem is to copy the content of the database to every device and to use a synchronization framework which propagates the changes from one devince to the others when there is a reliable network available.

This thesis presents a software library which can be included in mobile apps on different platforms and this software takes care of all the synchronization related tasks. The problems and solutions presented in this thesis can be separated into two areas: 1) How to build a software component which runs on all the major mobile operating systems (IOS, Android, and Windows). 2) How to synchronize the most commonly used database running on mobile devices (SQLite) with classical DBMSs.

The two topics are handled separately and the last part of the thesis combines all the findings and describes the implementation details of this framework.

© Copyright by Gergely Kalapos, 2015
All Rights Reserved

This thesis or any portion thereof
may not be reproduced or used in any manner whatsoever
without the express written permission of the author
except for the use of brief quotations in a review.

Contact: kalapos.azurewebsites.net

Contents

EIDESSTATTLICHE ERKLÄRUNG 2

STATUTORY DECLARATION 2

ABSTRACT 3
Contents 5

CONVENTIONS 7

1. INTRODUCTION 2
- 1.1 Problem description 3
- 1.2 The goal of this thesis 3
- 1.3 Requirements 4

2. EXISTING SOLUTIONS 6
- 2.1 Platform Independent Database Replication Solution Applied to Medical Information System 6
- 2.2 SQL server data replication 6
- 2.3 Oracle 20
  - 2.3.1 Oracle GoldenGate 21
- 2.4 Related technologies 26
  - 2.4.4 ORM Frameworks - Entity Framework 27
  - 2.4.5 ORM Frameworks - Apple Core Data 32
- 2.5 Summary 36

3 PROPOSED SOLUTION 37
- 3.1. SQLite 39
- 3.2 The SQLite type system 39
- 3.3. Calling C++ from a higher level language 42
- 3.4 Calling C++ from Objective-C 42
- 3.5 Wrapping C++ code into Windows Runtime Components 44
  - 3.5.1 Short Summary about C++/CX 47
- 3.6 Calling C++ from Java with the NDK 48
- 3.7 Summary 52

4. DETAILS OF THE PROPOSED SOLUTION FOR THE SYNCHRONIZATION PROBLEM 53
- 4.1 The initialization step 53
- 4.2 Change tracking between two synchronizations 55
- 4.3 Relationships between the tables 61
- 4.4 The Synchronization process 62

5 IMPLEMENTATION DETAILS 65
- 5.1 The architecture of the implemented system 65
- 5.2 C++11 move semantic in the DataTable and DataRow classes 76
- 5.2.1 Rvalues and rvalue references and the constructors of the DatabaseField class 80
- 5.3 Initial sync step with MS SQL Server 83

6. TESTING 84
6.1 Integration Testing 84
6.2 Sample application 86

7. SUMMARY 94
7.1 The API of MobileSync 94
7.1.1 The iOS API 94
7.1.2 The Windows API (C#, JavaScript, C++/CX) 99
7.2 Opportunities for further development 101
7.2 Conclusion 102

GLOSSARY 104

LIST OF FIGURES 108

REFERENCES 110

BIOGRAPHY 112

Conventions

In this document every source code uses the default coding standard and color scheme of the default editor of the given language. So for example every Objective-C code snippet uses the Apple coding guidelines and the default Xcode color scheme:

```
#import <Foundation/Foundation.h>
#import "IosDataTable.h"

@interface LocalDbManagerWrapper : NSObject

-(void)OpenDb;
-(void)CloseDb;
-(void)CreateTable:(IosDataTable*)withTable;
-(void)PerformInsert:(IosDataTable*)onTable andOnRow:(int)rowNumber;

@end
```

C# code uses Visual Studio color scheme and the standard Microsoft coding guidelines, Java uses the default coloring from Eclipse.

Since for C++ there is no general coding standard and also no default (or even de-facto) default editor I decided to use Xcode for the development. From this reason the chosen coloring and formatting schema for C++ is the default C++ coloring of Xcode. The reason behind this coloring and coding standard is to make the code samples in this document easier to read.

Beside that every class, primitive type, namespace, package, library, framework, database table and column name and in general everything which can be found in source code of the implementation and sample applications, or in the generated databases is written with italic letters.

I often use the terms “local database” and “remote database” in this document. These terms are meant from the perspective of the implemented framework, which runs inside mobile applications so “local database” refers to the database on the Smartphone/tablet pc and it is always a SQLite database. ”Remote database” refers to the database which is not on the phone and every time the framework accesses it a network connection needs to be established between the Smartphone/Tablet-PC and the database server.

One outcome of this thesis is a software component which solves the problems described here. This software component is called MobileSync, so every time the text refers to this name it means the framework which was implemented based on this thesis.

1. Introduction

One trend in today's information systems is the growing amount of data and the growing importance of mobile devices. An obvious consequence is that mobile devices are today very often an important part of a distributed information system beside the classical desktop PCs and laptops.

In business applications it is a common scenario that the system contains a central database and many client applications (including mobile apps) connect to this central database and fetch data on demand to show it on the User Interface immediately. Most of the systems today use an approach where it is assumed that the client devices are always online and they are able to send and receive data from the central database via different technologies like Web Services.

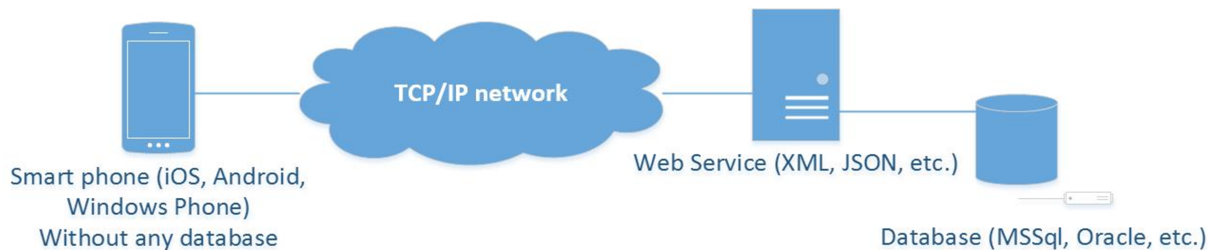


Figure 1: Classical Smartphone-Database communication

One main disadvantage of this approach is the assumption of a stable and fast TCP/IP connection between the mobile device and the server which hosts the data access layer.

There are still situations where these prerequisites cannot be fulfilled. One reason can be that the amount of data is too big and it is not possible to transmit everything on an unreliable wireless network. A second common reason is the lack of network infrastructure. There are many IT systems which are operating in extreme environments like in the middle of a desert. One example for these kinds of systems are logistic management systems for big and critical construction sites which are located far away from highly populated areas. These buildings are typically some kind of power plants or special factories. It is almost impossible to manage these kinds of construction projects without any IT system and a loosely coupled client-server infrastructure is predestined for this situation.

Of course there are many other possible use-cases where an “always online” approach does not work. Antoehr example would be an information system which operates in the mountains. Furthermore when a system is generally set up to support outdoor activities where clients are continously moving then the "always online" is again not the right way to go when the system is designed.

1.1 Problem description

This Master thesis focuses on the problem of data synchronization between mobile applications and classical database systems. One part of the problem is that mobile applications are written for different mobile platforms so supporting only one or two platform is not sufficient. The second part of the problem is the synchronization problem itself: on both sides we have relational databases, but these Database Management Systems follow completely different principles. A single database on a mobile device is typically used only by one mobile app and a server side database is normally used by multiple applications and users. This influences for example the type system of these database systems, so DBMSs on different sides use different type systems.

So the focus of this thesis can be split into two problem areas:

- How to synchronize a database which runs on a mobile device with another database which is run by a typical DBMS. Later parts of the thesis show that the DBMS on the mobile side is typically SQLite.
- How to design a software system (this software system is the synchronization framework itself) which can be embedded into applications running on multiple operating systems designed for mobile devices.

1.2 The goal of this thesis

There is no doubt that mobile devices like Smartphone and Tablet-PCs in the last years become very powerful. These devices can store as much data as classical servers only a few years ago and they also have enough processing power for complicated computations.

Therefore one reasonable approach to solve the issue described above would be to use the mobile devices not only as a thin client. We can store data in the 10 Gb range without any problem on a state of the art Smartphone or on a Tablet-PC. This means we can copy the content of a central database to the client devices and make the changes there when the device is not online. Later when it becomes online we can synchronize the changes back to the central database.

The goal of this master thesis is to examine the existing solutions in the field of data synchronization and to develop a solution for mobile platforms. At the end there should be a framework developed which can be included in mobile applications and this framework shall be able to handle all the work related to the synchronization. This framework can then enable to mobile application developers to build systems which work with a huge amount of data in environments where a service based solution cannot be used. The application can store all the data in a local database on the device which then can be synchronized with a

target database when the device comes online the next time. It is important that the framework shall be platform independent.

1.3 Requirements

One outcome of the thesis is an implementation of the discussed concepts. This section is a set of requirements for this implementation.

- The implemented system shall be able to **synchronize a database** running on a mobile device with a database running in a classical RDBMS.
- **Change tracking**: The framework shall be able to track changes between two synchronizations.
- **Multiplatform**: The system must run on IOS, Android and on Apps built on top of Windows Runtime (Windows Store Apps, Windows Phone Apps)
- **API fluidity**: The public API of the framework shall follow the typical API style of the given platform. For example when an IOS developer builds an application with the framework then interacting with the framework should be the same as interacting with any other IOS framework; when a Windows developer builds an application in C# then the framework should behave like a typical C# framework.
- **Automatic conflict resolution:** At least one solution shall be included in the framework to automatically resolve conflicts during synchronization.
- **Support for SQLite:** Later parts of the thesis show that the de facto standard RDBMS on mobile devices is SQLite. Therefore the system shall support SQLite as mobile side DBMS by default.
- Allow to **integrate** with new server side DBMSs
- **Simple public API**: The implemented framework is embedded in mobile Apps. These apps have their own code which interacts with the public API of the framework. This interaction should be as simple as possible in order to make the framework easy to use. Everything which is implementation details shall be hidden from user code. This includes change tracking, data conversion between the layers and memory management.
- The framework **does not change the data** stored in the database which it manages. This means that anything that a user code passes into the database remains unmodified when it goes through the framework. This is especially important with ids: the framework cannot modify an id, even if that id would cause conflicts during synchronization.

- **Minimal infrastructure:** The framework should work without installing any additional middleware software. The mobile app which includes the framework should be able to connect directly to the database.

2. Existing solutions

The first section of this chapter shows technologies and products which solve the problem described in the introduction. Common in these technologies and products is that they all focus on synchronization of different databases. The second part shows different Object-Relational-Mapping (ORM) frameworks, which are heavily related to the problem since a typical mobile app communicates through an ORM with the database.

2.1 Platform Independent Database Replication Solution Applied to Medical Information System

Tatjana Stankovic, Srebrenko Pesic, Dragan Jankovic, Petar Rajkovic presented a paper on the 14th East-European Conference on Advances in Databases and Information Systems (ADBIS 2010), held during September 20-24, 2010, in NoviSad, Serbia. The title of the paper is “Platform Independent Database Replication Solution Applied to Medical Information System” and they describe a solution which was used to replicate medical data between databases from Serbian health institutes. (Tatjana Stankovic, 2010)

The basic idea is that they don’t use change tracking tables and additional row identifier column like many other solution do, but they keep track of the performed SQL statements (like inserts, updates, deletes) and they store this history in a text file. At the synchronization step they send these files to the target database, do some processing and basically replay the activities on the remote side.

The biggest difference regarding requirements between the presented solution in that paper and in this thesis is that in the paper every database was a classical DBMS and no mobile devices were involved.

2.2 SQL server data replication

Microsoft introduced the data replication feature in SQL Server 6. The main goal of data replication is basically to copy data from one database to another. Of course there are additional capabilities included in the product, for example you can schedule the data replication, both one-way and bidirectional replication is supported, so one can configure a system also for complete data synchronization. This part of the thesis introduces the data replication features of SQL Server and its components and describes the different replication techniques that it offers. The technical description is based on the work of Sebastian Meine. (Sebastian Meine, 2013)

Components

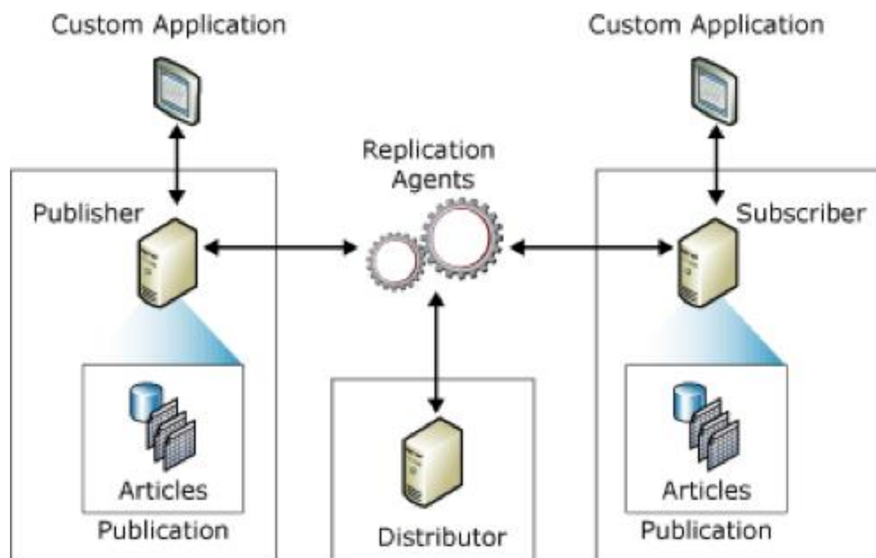


Figure 2: MS SQL Data Replication components (MSDN)

On the left side of the figure we see the **Publisher**. The publisher is a database server instance and within this instance there is a database which we want to replicate. The **Articles** are database objects inside the Publisher database and these articles contain the information which we would like to transfer to another location. An article is for example a table, a stored procedure or a view. An article also has properties which enable users to configure the replication. For example if the article is a table then it is possible to select some columns to replicate and leave all the other columns unsynchronized. A **publication** is a group of articles and it also has properties. One of the most important properties of a publication is the synchronization type, which basically defines which type of replication is used in this scenario. So for example if there are two tables in a database which must be replicated then two articles will be defined and grouped into one publication.

When you set up a replication sometimes you don't want to replicate the whole database. For this kind of scenarios the data replication provides filtering. There are two kinds of filtering options: horizontal filtering and vertical filtering.

When you select which articles you want to replicate you can select which columns on the table you want to synchronize. With this you can create horizontal filtering on the database.

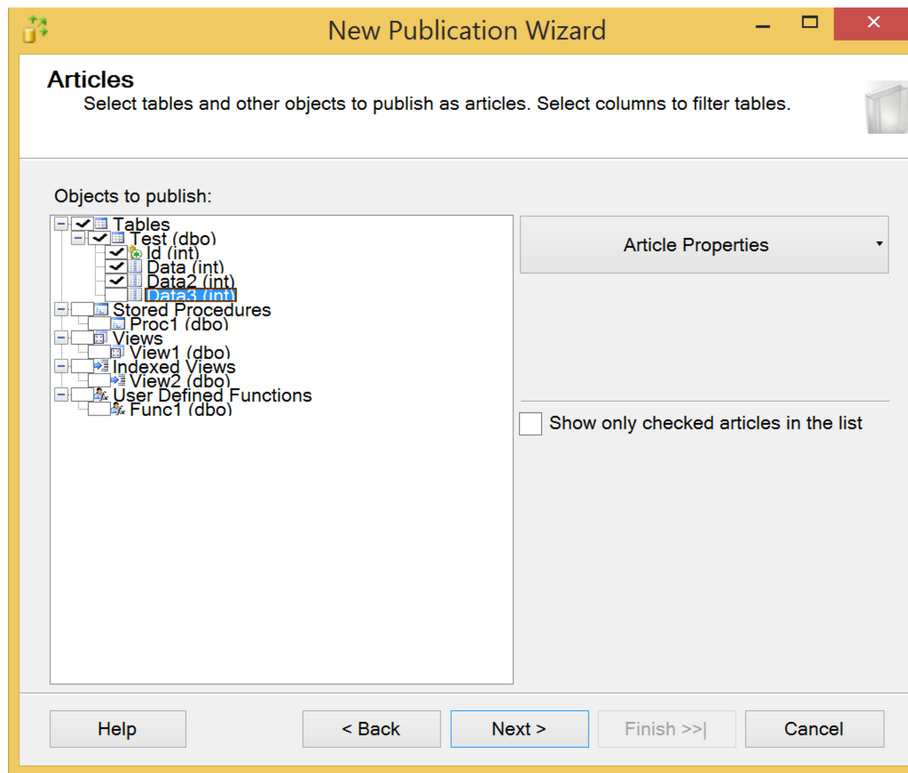


Figure 3: Selecting Articles in MS SQL Data replication

The tables by default replicate both the data and the schema changes in the table, although you can change this. Stored procedures behave similarly: Here you either propagate the definition of the stored procedure from the publisher to the subscriber, or you can choose to copy both the definition and every execution. Copying the execution means that every time a stored procedure is triggered on the publisher side it is also triggered during replication on the subscriber side with the same parameters. SQL server is smart enough to remember the modifications done by stored procedures and these changes are not synchronized. Views only replicate schema, since there is no underlying data stored in views.

During the wizard of the publisher there is another important page regarding filtering: the Filter Table Rows page. Here you can define different filter statements to keep some rows out of the synchronization process based on SQL statements. This is the vertical filtering of the replication.

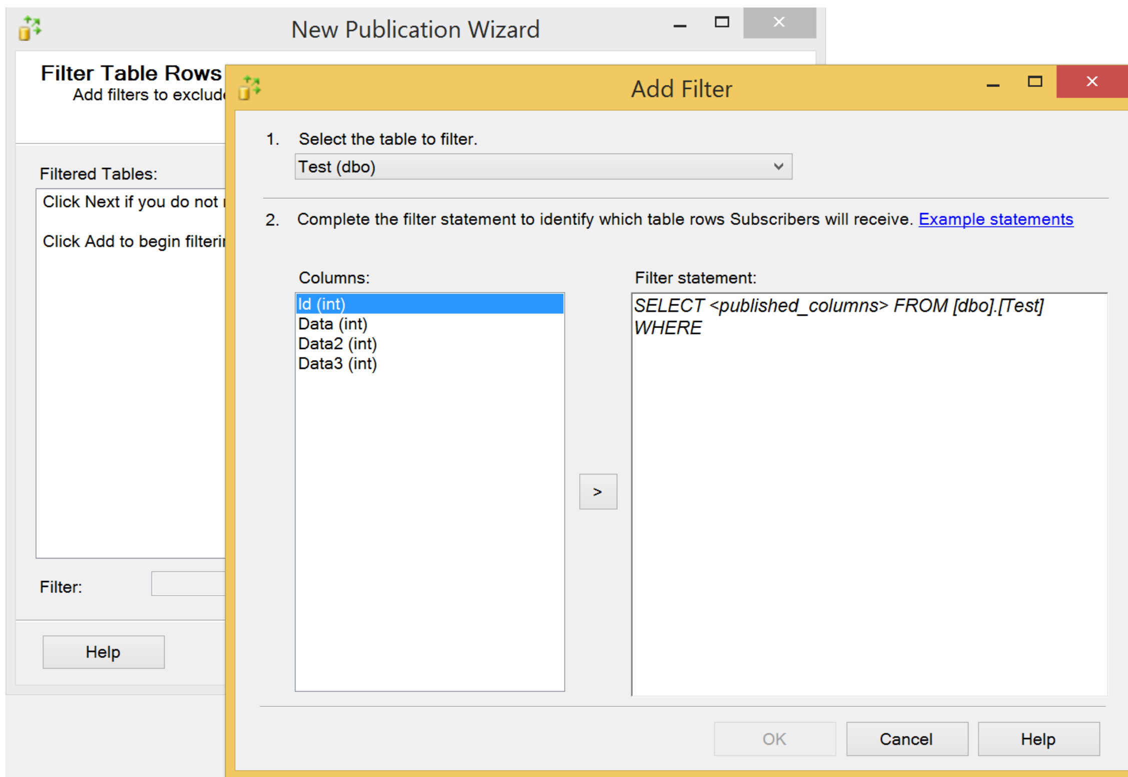


Figure 4: vertical filtering in MS SQL

In the middle of the Figure 2 there is the second component group of a data replication setup. The **distributor** is also a Database Server instance and it is responsible to identify the changes in the articles of the publisher database. Each publisher is connected to a distributor. The distributor can be configured to run on the same server as the publisher, although in a real world scenario this setup is very unusual. The replication agents are responsible for the execution of the replication process and by default the agents are executed by a SQL Server Agent job. There are different replication agents and they are responsible for the different steps of the replication process. Most of these agents run on the machine which hosts the distributor and they are normal windows executable which can be started independently from SQL Server, but of course the SQL Server Management Studio provides a UI for managing these agents. The main goal of the distributor is to keep the date available for subscribers. To define how long the data is kept you can define retention for every distributor. By retention we mean the time until the data is thrown away from the distributor. This is a very important concept, because 1) every subscriber must get the data, so if we have for example 5 subscriber in the system the distributor must wait until the last subscriber gets the date and 2) it is possible that a subscriber goes down forever or it is shut down, because it is not needed anymore. In this case the subscriber does not have to keep all the data forever since this would waste resources.

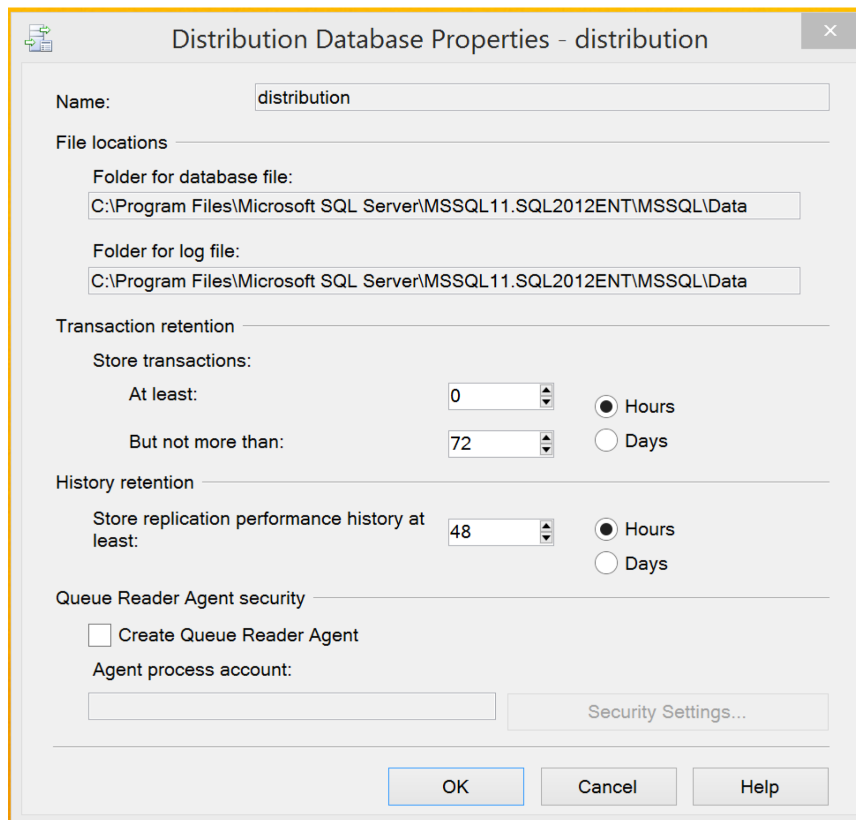


Figure 5: Distribution Database properties in MS SQL

To solve this problem you can define a maximum retention time for the database and if a subscriber does not synchronizes data until the maximum retention time reached the subscriber is marked as inactive and data on the distributor is cleared. When this subscriber comes online again it needs to be reinitialized.

On the right side of the Figure 2 we see the **Subscriber**, which is a SQL Server instance that contains a database which receives the data from the distributor. Each subscriber contains subscriptions which is the link between the publication and the subscriber. Depending on the configuration of the subscriptions it is either a push subscription of a pull subscription. As the name suggests with push subscription the Distributor is able to update the subscriber database and in the case of a pull subscription the subscriber database asks the distributor for data.

Although all of these components can be installed on the same machine, for a high performing system Microsoft recommends to provide for each component a dedicated machine. One common scenario for a smaller system is that the publisher and the distributor run on one machine and the subscriber on another one. When you set up a subscription the first thing you do is to select a publication to which the subscription connects.

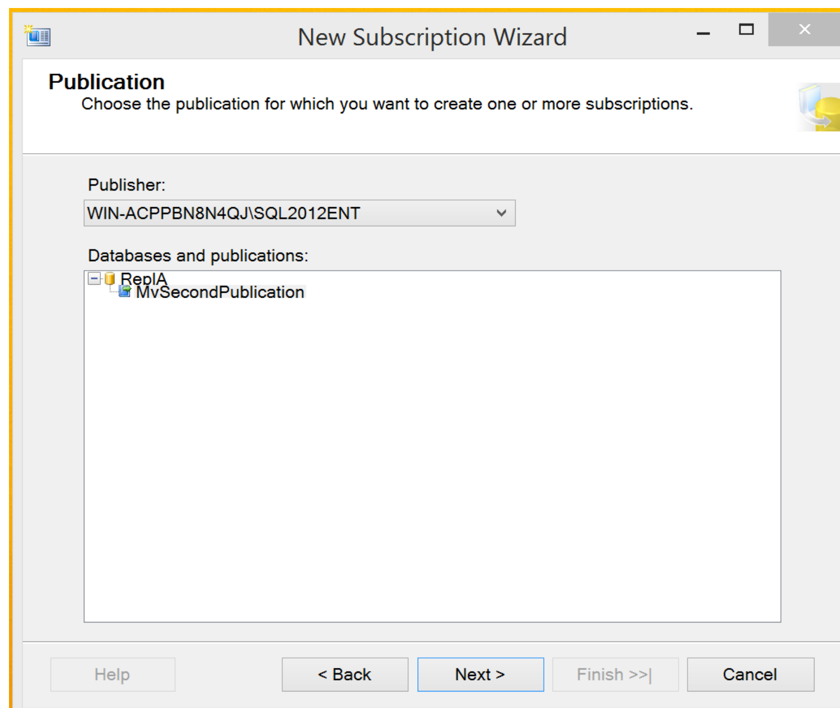


Figure 6: Adding Subscription in MS SQL.

The next step is to select the location of the distribution agent. With this you define either a push or a pull publication. When the distribution agent runs on the distributor it is able to push the changes to the subscriber, so this is the optimal choice when it is important that the changes are replicated from the one side to the other as quickly as possible. The second option is to install the distribution agent on the subscriber, so the replication process is started always by the subscriber, regardless whether data is changed or not. This set up is called the pull subscription. In this case the subscriber has full control on the data flow and this is the best choice in scenarios when the subscriber is not always online.

The next step is to select the subscription database, which can be any database on the subscriber, or you can also create a new database. One thing to note at this point is that the snapshot agent drops and recreates every database object which it would like to replicate. So for example if you chose the wrong database and it has tables or stored procedures with the same name then these objects will be dropped without any warning.

Types of replication

At this point all the important components were described, so we can move on to the different replication types.

1) Snapshot replication

The simplest type of replication in SQL server is the snapshot replication. In this case all articles and their data from the Distributor are copied to the Subscriber each time. This means that external changes in the Subscriber are overwritten and every time the complete

dataset is moved from the distributor to the subscriber. This can take up a lot of resources so the snapshot replication is rarely used in real world scenarios, but it is still important, because it serves as the initial step for other replication types.

2) Transactional replication

This type of replication uses log files to decide what data needs to be sent to the Subscriber. A change from the source database can be either immediately transmitted to enable almost real time synchronization between the parties or the replication also can be scheduled. The most important aspect is that transactional replication is sensitive to external changes and an external change can break the replication. Therefore the subscriber database must be considered as read-only in a default scenario. To use transactional replication every table must have a unique key.

In a transactional replication scenario there are 3 important agents involved: The snapshot agent, the Log Reader Agent and the Distribution agent.

The **snapshot agent** is (as the name suggests) also involved in a snapshot replication scenario. When it is used in a transactional replication setup it is always involved in the initial step: this agent creates a snapshot from the publisher database and stores it on the distributor. Although there are other ways to do the initialization of a transactional replication, the snapshot agent is the most commonly used initialization technique. The execution of this contains two main parts: first the snapshot agent creates a script which drops and creates the objects in the subscriber and in the second step it copies the content of the distributor database with the bcp tool into the snapshot folder of the distributor. It is very important to keep these snapshots up to date. Here we can have different scenarios: if we know that we will have only a limited number of subscribers and they are initialized immediately it is enough to run one snapshot and use this snapshot to initialize the subscribers. A different situation is when subscribers connect to the distributor in a rather unpredictable fashion: in this case it is better to schedule that the snapshot agent creates snapshots from the publisher database periodically to keep the content of the snapshot folder on the distributor up to date. This second scenario of course causes more loads to the publication database.

The **log reader** agent copies the log entries for every replicated database object from the publisher database to the distributor database. SQL Server uses logs to enforce ACID compliancy: for every change in the database in the first step a log entry is created and success of the change is only signed when the log entries are written to the disk. This way for example when the sever hosting the database goes down the logs can be used to finish the transactions which were started before the outage. The same mechanism is used for replication: the log reader agent searches for log entries to the replicated objects on the publisher and copies these entries to the distributor.

The **distribution agen**t runs either on the distributor or on the subscriber: in a push subscription it runs on the distributor, if it is a pull subscription it lives on the subscription. The task of this agent is to move data from the distributor to the subscriber. It always applies the changes from the distributor to the subscription in the order as they occurred within a publication. It is very important to keep in mind that the order across publications is not maintained, so if there is a foreign key relation between two tables which are contained in different publications then it can lead to foreign key violation. One other important thing to keep in mind is that by default a foreign key relation is not copied to the subscription database, so the scenario described before can run through without any warning, but it would then generate logical inconsistency in the subscription database.

3) Merge replication

This type of replication is designed to keep two or more databases in sync. So this time both databases can be changed and the merge replication keeps all the data in sync. This scenario allows that one row is updated in more than one database which can lead to conflicts. Merge replication ships with built in algorithms to automatically solve these conflicts. It uses a ROWGUIDCOL column which is set to "uniqueidentifier". If a table doesn't have such a column then it is automatically created during the configuration of the replication. Similar to the transactional replication this is also based on 3 important components: the 1) snapshot agent, 2) triggers, tables and views, and 3) the merge agent.

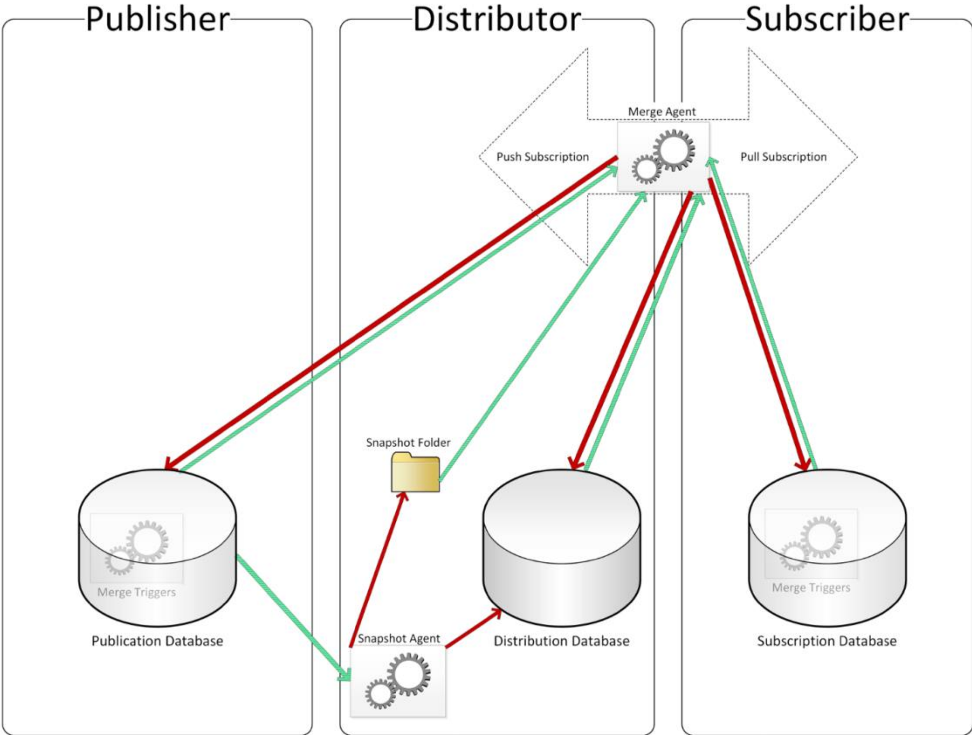


Figure 7: Communication between the MS SQL Data Replication components. Source: (Sebastian Meine, 2013)

The **snapshot agent** has the exact same purpose as in the transactional replication: it makes a snapshot from the publisher at a given point and this snapshot is used when a subscriber is initialized, and as with the transactional replication here we have also other possibilities to do the initialization step, but the snapshot agent is the most commonly used initialization. There is one difference regarding the snapshot agent in this scenario compared to the transactional setup: Because transactional replication uses the log reader agent in that scenario it is possible to track changes in the publication database during the snapshot is taken, so the snapshot agent does not need to take locks on the database. In a merge replication there is no log reader agent, so locks need to be taken to the database during snapshot: this means that during this time no one can write the database.

The **views, triggers and tables** have similar roles in this model as the log reader agent in the transactional replication, but they work completely differently: first of all since Merge Replication is bi-directional both the publisher database and the subscription database contain these additional objects. There are two types of triggers created: 1) triggers which track data changes in a table. There are three of them and each one follows the same naming pattern: MSmerge_???_HEXNumber. The HEXNumber is the internal article identifier and the ??? corresponds to the type of the trigger. There are 3 of them: del (the delete trigger), ins (the insert trigger), upd (the update trigger). 2) There are also triggers for schema changes: these are triggered when the structure of the database is changed: MSmerge_tr_alterschemaonly, MSmerge_tr_altertable, MSmerge_tr_altertrigger, MSmerge_tr_alterview.

When these triggers are called (or with other words when data is changed in the publisher database) the changes are stored in metadata tables.

The **Merge agent** in this case plays a similar role to the Distribution Agent: it propagates changes from the publisher to the subscriber, and (in contrast to the distribution agent) it also moves the changes from the subscriber to the publisher. It can run either on the Distributor (push subscription) or on the Subscriber (pull subscription). Merge replication is primarily designed for systems where some databases are often online and when they came online they must be synchronized with the rest of the system the pull subscription model is more popular in a merge replication setup. This also has the advantages that it reduces the load from the Distributor, since the merge agent runs on the Subscriber.

To create a publication for merge replication the same wizard can be used as with other replication type. The difference is that for the publication on the Publication type page you select merge replication.

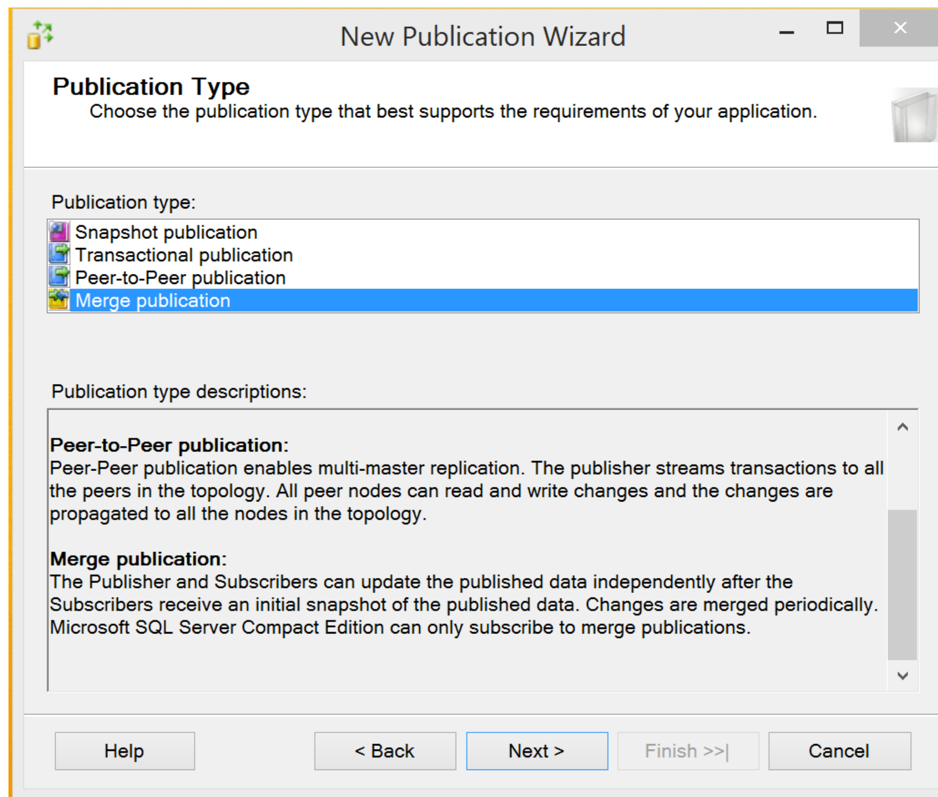


Figure 8: Merge replication

One difference regarding subscriptions between the two types of replications is that merge replication tracks schema only changes for articles, except for tables: so for example it is not possible to set up a merge replication in the way that when a stored procedure is called on the publication then the stored procedure is called with the same values on the subscription. The reason for this is that transactional replication reads logs and in that case these kinds of stored procedure calls can be replayed very easily from the logs, whereas the triggers and additional tables are not suitable for such tasks. Another difference is that merge replication is able to replicate tables without foreign keys (this is not the case in transactional replication).

During the setup you can also select the Synchronization direction: it can be 1) Bi-directional (changes can happen on both side), 2) Download-only to Subscriber, which allows only subscriber changes or 3) Download-only to Subscriber, which prohibits Subscriber changes. This option has also significant impact on the performance, since the Download-only with prohibit subscriber changes option does not need any conflict resolution information, since in that case no conflict can occur. Another important article property is the type of the resolver: in a bi-directional replication conflicts can occur, meaning that data is changed on both sides on the same artifact. These conflicts can be update-update conflicts where both changes are updates, update-delete conflict where on the one side for example a row is updated and on the other side the same row was deleted. There is a so called failed-change conflict which refers to cases where one change on one side of the replication violates

constraints on the other side of the database. A simple example for this scenario is where with one change we violate a foreign key relation, but this only leads to problems during synchronization. SQL Server provides so called conflict resolvers that can be selected for every article to resolve conflicts. There are many different resolvers already installed in the product, for example the Maximum Conflict resolver takes a column name as parameter and based on the value of this column decides which side wins: the row with the lower value will be dropped.

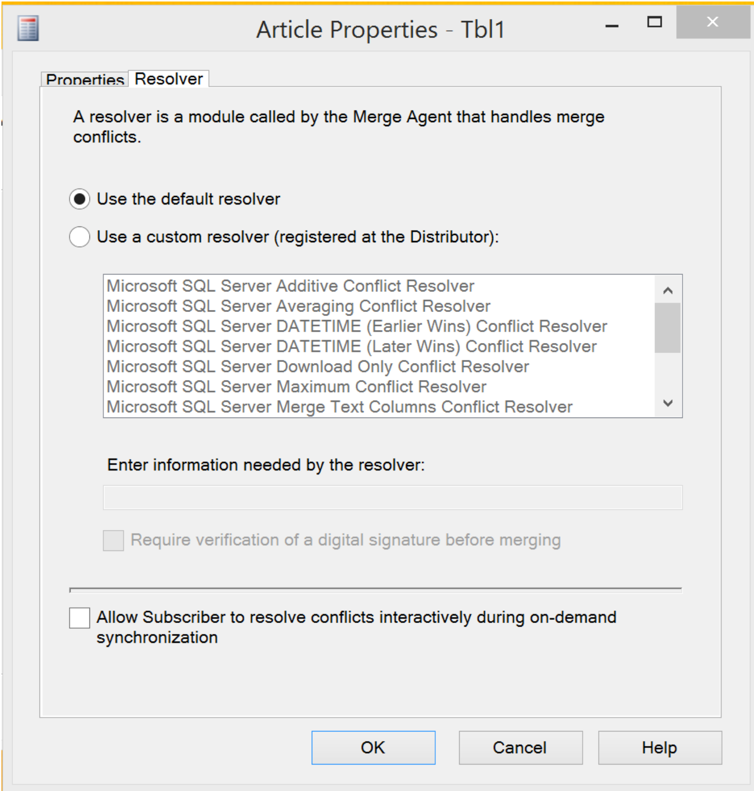


Figure 9: SQL Server conflict resolver

The default resolver uses the priority levels to decide which side wins in a conflict. SQL server also provides many options to customize or to plug in your own conflict resolver. For example the Stored Procedure resolver uses a stored procedure to select the winner row, which is the easiest way to write a custom resolver. Custom resolvers can also be created with C++ or even in managed languages which run on the Common Language Runtime (CLR). This latter option is called Business Logic Handlers and these are the most flexible resolvers. They can not only select the winner row, but also completely reject a change or even modify the values.

When you run synchronization on a subscription you can see a summary which show the number of changes and conflicts after the synchronization is finished.

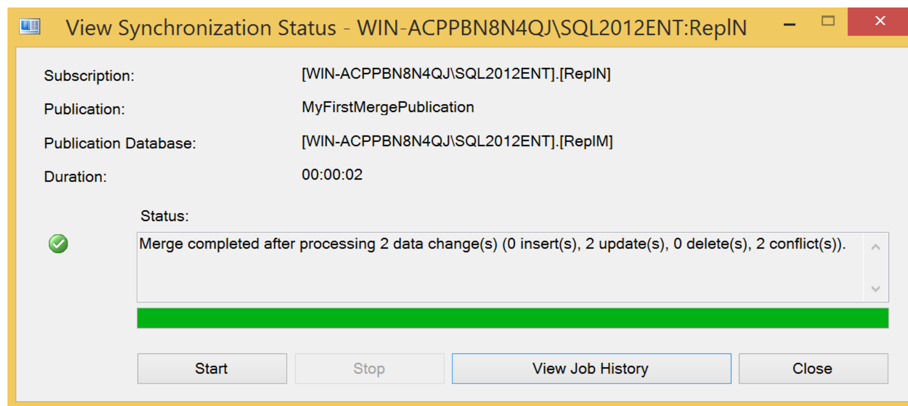


Figure 10: MS SQL synchronization summary

On the publisher side you can also see the details of the resolved conflicts and you can also change the values afterwards.

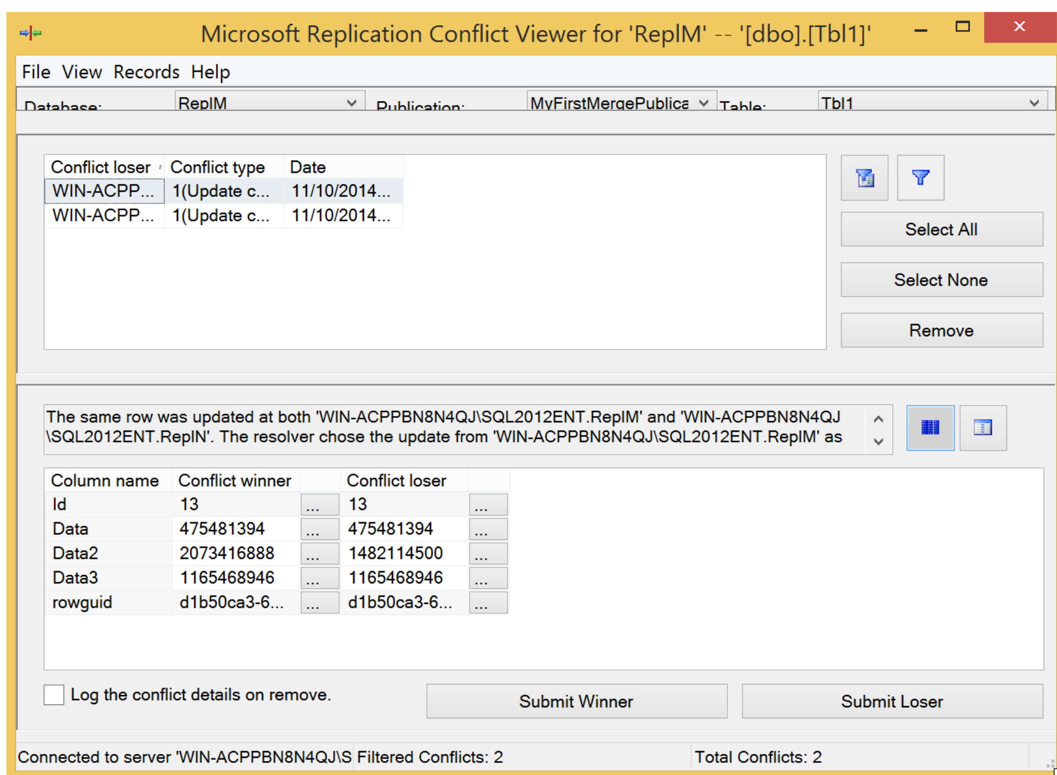


Figure 11: MS SQL detailed view of the resolved conflicts

The next important option is the tracking level. SQL Server provides two types of tracking levels: 1) row-level tracking and 2) column-level tracking. The default is row-level tracking and it means that the change tracking handles a complete row as one unit and it is identified by the uniqueidentifier column. This means that if one or more column is changed in a row then the row is registered with its identifier as a changed row, but nothing more is saved

regarding the change, so the same information is created when a single column is changed and when all the columns are changed. This can mark changes as conflicts which are not really conflicts: for example if column C1 is changed on one side and column C2 is changed on the other side. The advantage of this mode is that is requires less resources. The column-level tracking identifies not only a row, but it also registeres which column was changed. This means that only real conflicts are marked and conflicts, but of course this requires more resources.

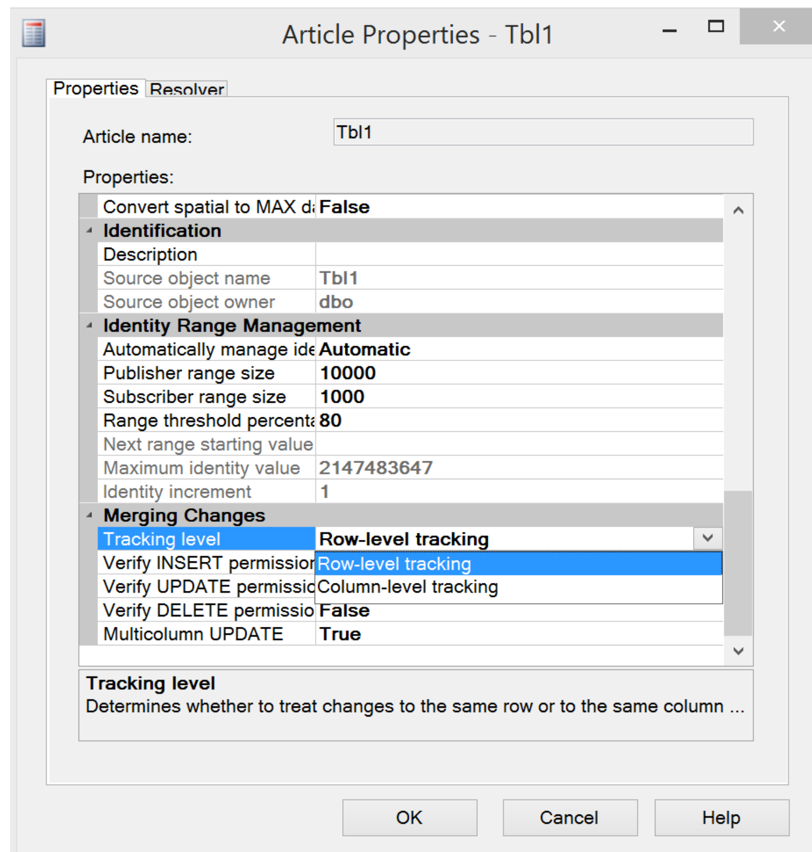


Figure 12: MS SQL Tracking level

As soon as you set up the subscription an additional uniqueidentifier column will be added to all the tables. When you turn off the replication these columns will be removed, so it is very important to keep in mind that these are not part of the database logic, they are there for data replication, so every code which relies on this can break when someone turns off the replication.

When you set up a subscription in merge replication you have to choose the subscription type of the new subscription. Here you have two options: 1) Client means that its publication acts as a client and it does not propagate changes to other databases, except the publication database in a bi-directional setup. 2) Server publication means that you can define sub-subscriptions for the subscriber and it can act as a publisher to other subscriptions. For

server subscribers you can also configure the priority level which is used by the default conflict resolver to decide which subscriber wins in a conflict. Server with higher priority wins over servers with lower priority.

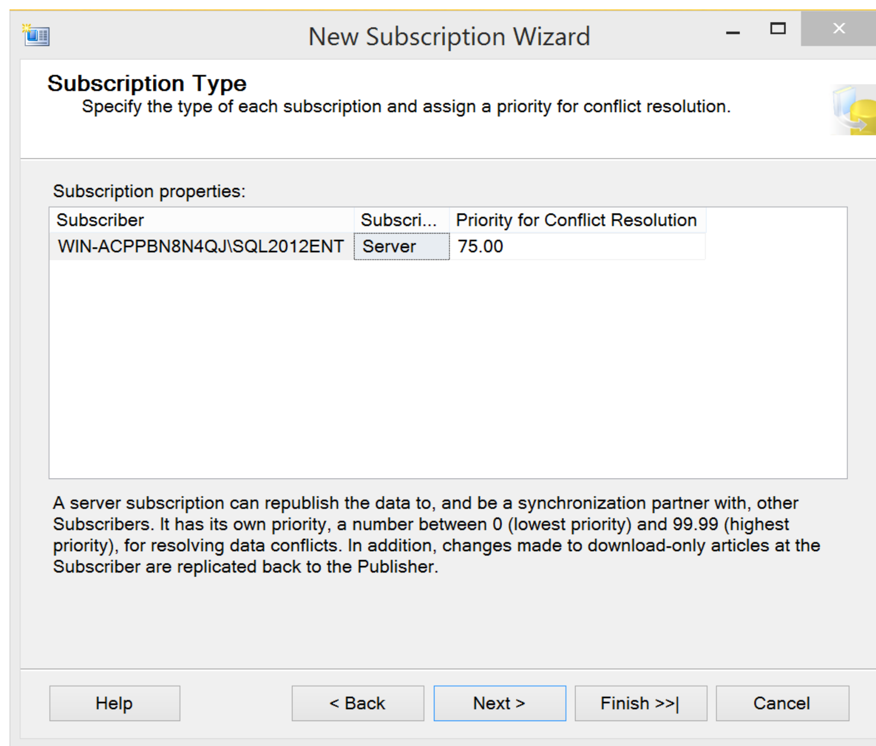


Figure 13: MS SQL subscription type

When a merge subscription is configured you can trigger synchronization manually any time by right clicking to the Subscription under Replication->Local Subscription in SSMS and by choosing View Synchronization Status. Here a new window comes up where by pressing the start button you start the merge agent which triggers the replication process.

Security Issues

The data replication feature of MS SQL also provides many options the increase the security of a replicated system. For example when you set up a transactional replication the access rights of the snapshot folder can be also maintained by the system administrator meaning that if a distributor or a subscriber does not have the appropriate access rights to this folder they will never see data from the publisher database.

Another important security measure built into the system is the Publication Access List (PAL). The goal of the PAL is to prevent unauthorized users to access publication data. This list exists independently from the other SQL Server permissions and it contains the authorized users for data replication.

Every login which is added to the PAL must exist both on the Distributor and on the Publisher and the user associated with the login must exist in the publication database.

The PAL is stored in the publication database; this is the reason why the user must exist in that database.

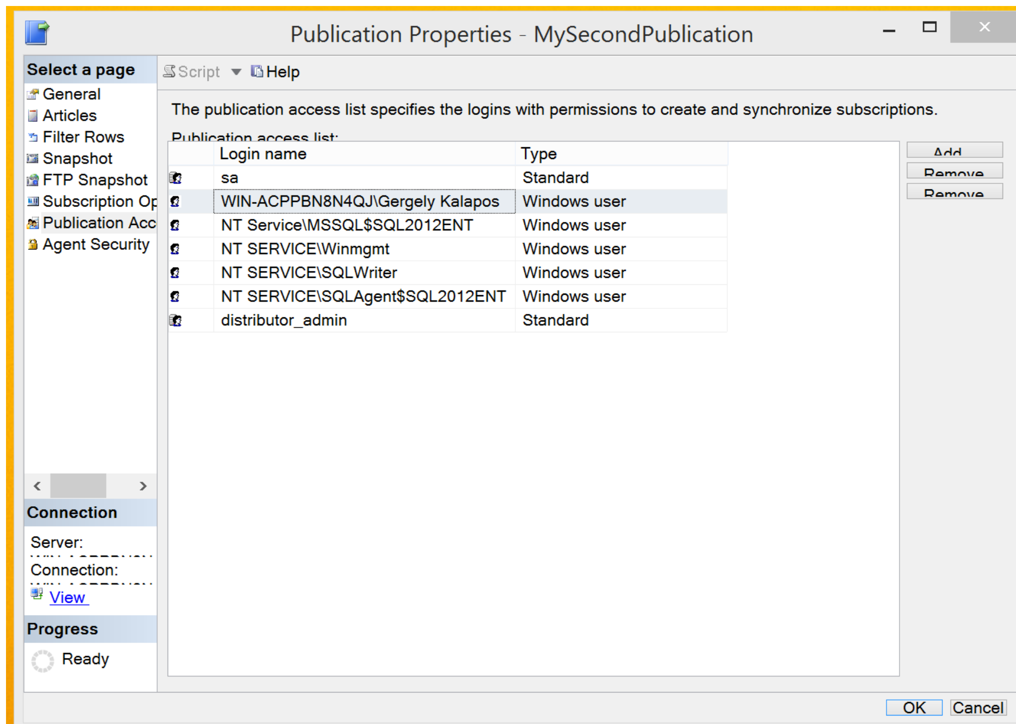


Figure 14: MS SQL Security Properties

One related feature to data replication is the Always-On feature, which was introduced in SQL Server 2012. With Always-On you can the database server automatically keeps a second database in sync with the master database to enable high availability (HA). This enables read-only access to the second database, so when the master database goes down the data is still available for the clients. Before SQL Server 2012 to enable such a functionality for a database data replication was the way to go, but with that version for these scenarios Microsoft advices to use the Always-On features. But there are still many scenarios which can be solved only with data replication, since it has more flexibility (syncing only a subset of the tables, fill out rows, sync from multiple databases into one, etc.)

2.3 Oracle

Oracle offers multiple solutions to synchronize databases. One of this solutions is called Oracle streams which enables users to synchronize databases both in homogenous and in heterogeneous environments.

Oracle streams define two types of database: the source database is where the data comes from and the destination database if where the data will be propagated. To reach this goal Oracle streams executes three steps:

1) In the capture process it collects logical change records (LCR) which basically describes DML and DDL changes in the target database

2) In order to move the LCRs from the source to the target database Oracle streams uses queues. A queue can be directly on the machine where the target database resides, but it can also be an intermediate queue.

3) When the LCRs arrive to the destination Oracle applies the LCRs to the target database.

The first versions of Oracle streams was shipped with Oracle 9i and today it is only part of the Enterprise Edition until version 11gR2. In July 2009 Oracle acquired Golden Gate, a company specialized in real time data integration solution and since version 11gR2 Oracle recommends to use GoldenGate for data synchronization. Therefore the remaining part of this chapter focuses on Oracle Golden Gate.

Source: (Oracle FAQs) (Oracle Streams)

2.3.1 Oracle GoldenGate

This section introduces the main components of Oracle GoldenGate and some of their configuration options. The goal here is to give the reader an overview about the product. This section is based on (Gupta, 2013).

Oracle Golden Gate is a software product for real-time data integration and transactional replication for heterogeneous data systems. Beside Oracle database it supports many other DMBSs, for example MySQL, DB2, SQL Server and it runs on many different platforms.

One of the key components of the software is the so called “Capture" which runs external to the DBMS and it captures changes in the database. The capture component produces Trail files which are used to queue transactions to route them to other systems.

The Golden Gate pump process is the component which is responsible for the transmission of the trail files from one system to another. The trail files are transmitted via TCP/IP.

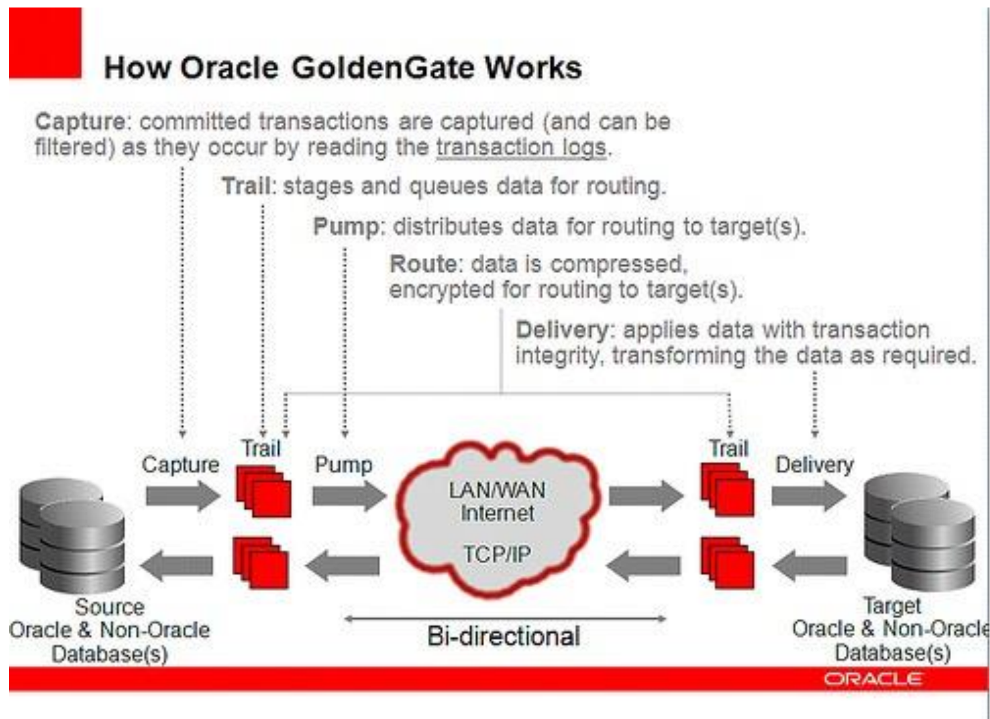


Figure 15: Oracle GoldenGate components (Gupta, 2013)

The delivery module is installed on the target system and its purpose is to move the data into the target database.

The same components can be installed to support the whole process in the opposite direction. With this a bi-directional synchronization can be configured. Beside Unidirectional and Bidirectional setup Oracle Golden Gate supports multiple Deployment models:

- Unidirectional
- Bidirectional
- Peer-to-Peer
- Broadcast
- Integration/Consolidation
- Data distribution

In order to set up GoldenGate for a database first you have to enable supplemental login for the source database with the following SQL commands:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;

ALTER DATABASE SWITCH LOGFILE;
```

In the source database a user must be created with special privileges. Here are the command which are needed for the new user (In this example the user for the GoldenGate processes is called GGATE_ADMIN)

```
GRANT CREATE SESSION, ALTER SESSION to GGATE_ADMIN;
GRANT ALTER SYSTEM TO GGATE_ADMIN;
GRANT CONNECT, RESOURCE to GGATE_ADMIN;
GRANT SELECT ANY DICTIONARY to GGATE_ADMIN;
GRANT FLASHBACK ANY TABLE to GGATE_ADMIN;
GRANT SELECT ANY TABLE TO GGATE_ADMIN;
GRANT SELECT ON DBA_CLUSTERS TO GGATE_ADMIN;
GRANT EXECUTE ON DBMS_FLASHBACK TO GGATE_ADMIN;
GRANT SELECT ANY TRANSACTION To GGATE_ADMIN
```

After the permissions are granted to the selected user the logging can be enabled in GoldenGate with the **GGSCI** command. This can be done with the ADD TRANDATA command where the parameter is the name of the table for which logging is turned on. To turn on logging for all the tables in the schema the '*' sign can be used.

```
Select Oracle GoldenGate Command Interpreter for Oracle

GGSCI (WIN-ACPPBN8N4QJ as system@orcl/CDB$ROOT) 16> ADD TRANDATA CDB$ROOT.SYSTEM
.SYNCTEST
This operation will modify an object at the root level of a consolidated databas
e, continue? (Y/N): y

2015-01-04 18:51:17  WARNING OGG-06439  No unique key is defined for table SYNCT
EST. All viable columns will be used to represent the key, but may not guarantee
 uniqueness. KEYCOLS may be used to define the key.

Logging of supplemental redo data enabled for table CDB$ROOT.SYSTEM.SYNCTEST.
TRANDATA for scheduling columns has been added on table 'CDB$ROOT.SYSTEM.SYNCTES
T'.
GGSCI (WIN-ACPPBN8N4QJ as system@orcl/CDB$ROOT) 17>
```

Figure 16: Oracle setting up logging for GoldenGate

The **manager process** is the root of the GoldenGate instance and this component must be configured both on the target and on the source side. It needs a configuration file which can be done with the Edit params MGR command. This command opens an editor and the configuration file of the manager. Here we configure timing settings and the location of the trail files. With the Start mgr command the manager is started.

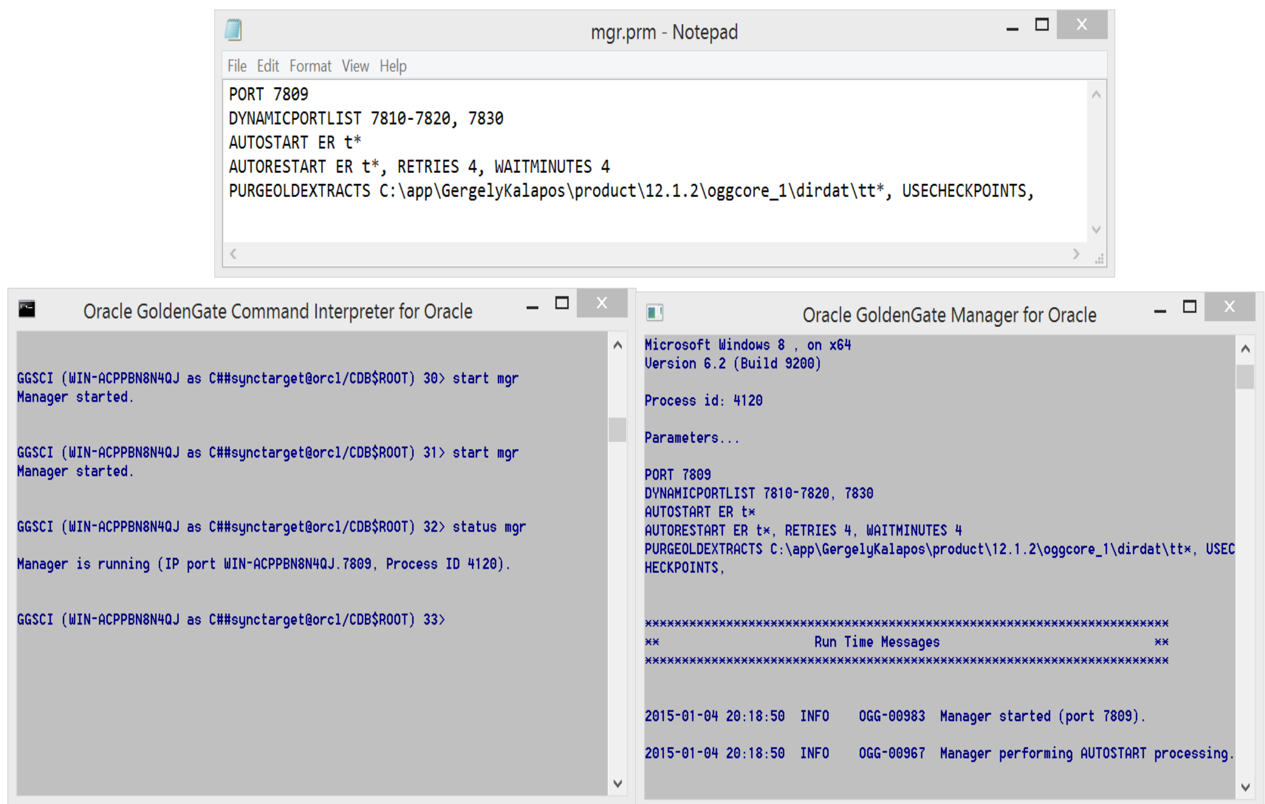


Figure 17: The GoldenGate Manager process

The next GoldenGate component which must be configured on the source side is the **extract component**. This is the component which produces the trail files and it can run in two modes:

1) The classical capture mode accesses the redo logs of the database. The only prerequisite is that the extract process needs access to the logs. The disadvantage is that some of the data types are not supported in this mode.

2) The integrated capture mode was introduced in Version 12.2.0.3. In this mode the extract process works directly with the database log mining server and it is able to receive LCRs (introduced by Oracle streams). Oracle 12c introduced the new multi-tenant architectures with pluggable databases. In this case every pluggable database belong to one so called "container database" and this way they share one redo streams. In this case GoldenGate has to be able to filter out the logs which do not belong to the database which is selected for synchronization and this only works in integrated mode. So Oracle 12c (the newest version at the point of writing) only supports integrated mode.

In order to configure the extract process we need a configuration file (similar to the manager). The capture mode is defined in this configuration file. An example for the classical capture mode configuration file can be seen in the following code snippet:

```
EXTRACT EGGTEST1

USERID GGATE_ADMIN@orcl, PASSWORD GGATE_ADMIN
```

```
EXTTRAIL C:\app\GergelyKalapos\product\12.1.2\oggcore_1\dirdat\st
TABLE system.synctest;
```

Here only the system.synctest file is defined for capturing, but the "*" syntax can here also be used to capture the whole schema.

You start the extract process with the START EXTRACT <Name> command.

Figure 18: GoldenGate capture process

An optional process that can be configured is the **data dump process** which always resides on the source system. If it is not configured then the extract component transfers the data to the target system, otherwise the data dump process takes care of the transferring.

A data dump process is basically a secondary extract process and you configure it to become a data dump process in the configuration file by adding the RMTHOST <HOSTNAME_IP_TARGET_SYSTEM>, MGRPORT <TARGET_MGRPORT> line.

The **replicat process** runs on the target system and its job is to propagate the content of the trait files into the target database. It is also responsible for the transformation of the data if any mapping transformation is defined.

The first step to set up the replicat process is to create a checkpoint table with the ADD CHECKPOINTTABLE <SCHEMA.TABLE> command. In this table GoldenGate creates checkpoints which are used for example when GoldenGate unexpectedly exits to pick up the synchronization process at the same point where the process was terminated.

After the configuration is done the replicat can be started with the START REPLICAT <REPLICAT> process.

By starting all the components introduced in this section a basic replication between two databases can be already configured.

As a summary here are the components which need to be configured for this basic setup:

Source side:

- Admin user added to the source database
- GoldenGate manager process must be configured and started
- Extract and DataDump process must be configured and started

Target side:

- Admin user added to the target database
- GoldenGate manager process must be configured and started
- Checkpoint table created to the Replicat process
- Replicat process must be configured and started

There is obviously much more to say about this topic, for example you can set up multiple instances of each component (which is the case in many real-life scenarios) and you can define different mappings between the tables. The goal was here to introduce the main components and features of Oracle GoldenGate.

2.4 Related technologies

There are technologies which are used for similar purposes as MobileSync, but they do not have the exact same functionality. One aspect of MobileSync is that it keeps track of every change. This framework is embedded into Mobile Apps, so it means that the data access layer of these Apps must inform MobileSync about the changes. To solve this there are two different approaches: 1) the framework must be integrated with the existing data access frameworks or 2) the framework itself must provide the data access layer for App Developers.

This section describes two related data access frameworks. The goal here is to show how an application typically accesses a database today.

Later sections go into design decisions and implementation details and these later sections will show how data access works with MobileSync.

2.4.4 ORM Frameworks - Entity Framework

Entity Framework (EF) is an open source object-relational mapping from Microsoft which is the most used ORM framework on the .NET platform. The goal of an ORM framework to map relational database entities like tables, rows and column into object oriented programming concepts like collections, classes and class member fields (or as it is called in C# properties). This section is based on (Christian Nagel, 2014)

Although EF does not solve the synchronization problem itself, it still has two important concepts which are related to MobileSync: 1) it is a data access layer and it generates SQL commands from user code. MobileSync also has to offer public APIs, so that every database change from the user code also triggers the change tracking code. 2) The framework itself has change tracking in itself: until the changes are not saved by the user code they are kept in memory in the application layer, so when the application changes something in the memory the changes are persisted to the database only when the application calls the *SaveChanges* method.

It defines three different programming models:

- Code first: in this case the initial stage is that application developers wrote model classes and based on these model classes EF generates the database schema automatically. Since this programming model is outside the scope of the problem description the thesis does not go further into the details of code first programming model in Entity Framework.
- Database first: here the initial stage is an existing database which must be connected to a .NET application. With the database first approach the framework looks at the database schema and based on this database schema it creates C# classes with properties. One instance typically maps to one row in a table and every property on this instance is the value of a field in that table. This one-to-one mapping is not the only possible mapping style; EF also supports different mappings, where for example two tables are represented by a single C# class.
- Model first: in this scenario first you create a model in the model designer and based on this model the system creates both the database in the RDBMS and the C# classes on the application side.

The code first approach only needs the C# classes to create the database and the mapping, the Database first and Model first approaches need 3 layers to be defined:

- Logical: the definition of the relational data
- Conceptual: the definition of the .NET classes

- Mapping: defines the relationships between the logical and the conceptual layer.

The logical layer is defined by the Store Schema Definition Language (SDL) and describes the structure of the database tables and the relationships between them. The following exampe shows a section of the Northwind database described in the SDL language.

```
<edmx:StorageModels>

  <Schema Namespace="NORTHWNDModel.Store" Provider="System.Data.SqlClient"
      ProviderManifestToken="2012" Alias="Self"
      xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
      xmlns:customannotation="http://schemas.microsoft.com/ado/2013/11/edm/customannotation"
      xmlns="http://schemas.microsoft.com/ado/2009/11/edm/ssdl">

    <EntityType Name="Categories">

      <Key>

        <PropertyRef Name="CategoryID" />

      </Key>

      <Property Name="CategoryID" Type="int" StoreGeneratedPattern="Identity" Nullable="false" />

      <Property Name="CategoryName" Type="nvarchar" MaxLength="15" Nullable="false" />

      <Property Name="Description" Type="ntext" />

      <Property Name="Picture" Type="image" />

      <Property Name="RowGUID" Type="uniqueidentifier" />

    </EntityType>

 <EntityType Name="Categories">

        …

</edmx:StorageModels>
```

The conceptual layer is defined by the Conceptual Schema Definition Language (CSDL). This sample also uses the Northwind database.

```
<edmx:ConceptualModels>

    <Schema Namespace="NORTHWNDModel" Alias="Self" annotation:UseStrongSpatialTypes="false"
xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
xmlns:customannotation="http://schemas.microsoft.com/ado/2013/11/edm/customannotation"
xmlns="http://schemas.microsoft.com/ado/2009/11/edm">

    <EntityType Name="Category">
```

```
    <Key>
      <PropertyRef Name="CategoryID" />
    </Key>
    <Property Name="CategoryID" Type="Int32" Nullable="false" annotation:StoreGeneratedPattern="Identity"
                                                                                                    />
    <Property Name="CategoryName" Type="String" MaxLength="15" FixedLength="false" Unicode="true"
                                                                                      Nullable="false" />
    <Property Name="Description" Type="String" MaxLength="Max" FixedLength="false" Unicode="true" />
    <Property Name="Picture" Type="Binary" MaxLength="Max" FixedLength="false" />
    <Property Name="RowGUID" Type="Guid" />
    <NavigationProperty Name="Products" Relationship="Self.FK_Products_Categories" FromRole="Categories"
                                                                                      ToRole="Products" />
  </EntityType>
     …
</edmx:ConceptualModels>
```

And finally the Mapping uses the Mapping Specification Language (MSL):

```
 <edmx:Mappings>
  <Mapping Space="C-S" xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
   <EntityContainerMapping StorageEntityContainer="NORTHWNDModelStoreContainer"
                                        CdmEntityContainer="NORTHWNDEntities">
    <EntitySetMapping Name="Categories">
     <EntityTypeMapping TypeName="NORTHWNDModel.Category">
      <MappingFragment StoreEntitySet="Categories">
       <ScalarProperty Name="CategoryID" ColumnName="CategoryID" />
       <ScalarProperty Name="CategoryName" ColumnName="CategoryName" />
       <ScalarProperty Name="Description" ColumnName="Description" />
       <ScalarProperty Name="Picture" ColumnName="Picture" />
       <ScalarProperty Name="RowGUID" ColumnName="RowGUID" />
      </MappingFragment>
     </EntityTypeMapping>
    </EntitySetMapping>
```

```
        …
  </EntityContainerMapping>
 </Mapping>
</edmx:Mappings>
```

The 3 layers are defined in an EDMX file and in case of a database first model it is created by a wizard. Visual Studio provides a graphical editor for this file, but by opening it in a text editor you can see end edit the layers by using SSDL, CSDL and MSL directly.

For .NET applications connecting to a database a so called connection string is stored in a configuration file. For Entity Framework the connection string contains a special part, which points to the EDMX file where the three layers are defined. For example here the northward database and the NorthwindModel.edmx are referenced.

```
<connectionStrings>
    <add name="NORTHWNDEntities"
       connectionString="metadata=res://*/NorthwindModel.csdl|res://*/NorthwindModel.ssdl|r
       es://*/NorthwindModel.msl;provider=System.Data.SqlClient;provider connection
       string=&quot;data source=WIN-ACPPBN8N4QJ\SQL2012ENT;initial
       catalog=NORTHWND;integrated
       security=True;MultipleActiveResultSets=True;App=EntityFramework&quot;"
       providerName="System.Data.EntityClient" />
  </connectionStrings>
```

In order to access the database you need to use the data context created by EF. This class is created by the designer and it is always a subclass of the *DbContext* type and it is basically the connection point from the .NET application to the database.

To querying the database Entity Framework provides many options. One option is to use Entity SQL which an extension of the T-SQL language to query a database via Entity Framework.

The disadvantage of Entity SQL is that it uses strings instead of strongly typed C#/VC classes. This is the example how you set up a select query.

```
var connection = new EntityConnection(connectionString);
await connection.OpenAsync();

EntityCommand command = connection.CreateCommand();
command.CommandText = "[NorthwindModel][Category]";

DbDataReader reader = await command.ExecuteReaderAsync();
```

The most commonly used technique to query a database with EF is by using LINQ. LINQ stands for Language Integrated Query and it is a part of the C# language. It allows C# developers to write SQL like syntax when they deal with data. LINQ can also be used with other technologies, not only with

EF. In fact with LINQ to object one can manipulate normal .NET Lists and other collections with the same syntax.

Here is an example how to query a database with LINQ to EF:

```
var categories = from item in context.Categories
                 select item.CategoryName;

string firstItemName = categories.First();
```

Here are no strings involved, so for example when the database schema is defined and the edmx model is updated to the new schema then the properties on the Categories class are completely changed and the compiler throws an error at compilation time. With this application developers can avoid bugs at the data access layer.

The data context class also supports object tracking. This means that if two queries return the same database object (for example the same row from the database) then they return the same entity object and by changing the first result the second one is also automatically changed.

As the introduction of this section mentioned EF is also able to check changes. This means when an item in C# is changed then the data context instance keeps track of this changes.

For example let's examine the following code snippet:

```
var category = (from item in context.Categories

                select item).First();

 category.CategoryName = "changed";

 foreach (var item in context.ChangeTracker.Entries<Category>())
 {
     if (item.State == System.Data.Entity.EntityState.Modified)
     {
         Console.WriteLine("Original:");
         foreach (var originalValProp in
               item.OriginalValues.PropertyNames)
         {
             Console.WriteLine(item.OriginalValues[originalValProp]);
         }

         Console.WriteLine("New:");
         foreach (var newValProp in item.CurrentValues.PropertyNames)
         {
             Console.WriteLine(item.CurrentValues[newValProp]);
         }

     }
 }
```

Here we select the first category and change the category name to “changed”. By iterating through the Entries property of the ChangeTracker of the context we get the items which were changed. The output of the code on the Northwind database is the following (the value "Beverages" is changed to the string "changed":

Original:

1

Beverages

Soft drinks, coffees, teas, beers, and ales

System.Byte[]

5941de03-8542-4c10-9f9d-5065fa54509d

New:

1

changed

Soft drinks, coffees, teas, beers, and ales

System.Byte[]

5941de03-8542-4c10-9f9d-5065fa54509d

At this point in the database we still have the old value. By calling the *SaveChanged* method on the data context the changes are propagated to the database. This change tracking functionality can be turned off by calling the AsNoTracking method on a result set.

2.4.5 ORM Frameworks - Apple Core Data

Based on (Apple inc.) (Hegarty, 2013-14 Fall Semester, Lecture 13)

Apple Core Data serves a similar purpose on Apple platforms as Entity Framework on Microsoft platforms: it allows application developers to query a database with statically typed OOP classes. Since on iOS the default database is SQLite, core data also uses it as its default data storage, but it can also be configured to store data in XML files.

As Apple describes it its main purpose is to “provide generalized and automated solutions to common tasks associated with object life-cycle and object graph management, including persistence” (Apple inc.). It has support for undo and redo changes, automatic relationship management with the dot notation, schema migration and data persistence, and lazy loading to minimize the memory footprint.

One of the reasons why Apple suggests using Core Data is that it significantly reduces the number of lines needed for a typical data persistence scenario (the code with Core Data is typically 50-70% smaller then the code for the same task without Core Data). (Apple inc.)

Another fact about Core Data is that it is very well integrated into Xcode: it can generate statically typed classes based on the mappings defined on the graphical interface.

The first step to use CoreData is to create a visual mapping between the database and the Objective-C classes. For this Xcode offers a graphical interface and it stores the data in an xcdatamodeld file.

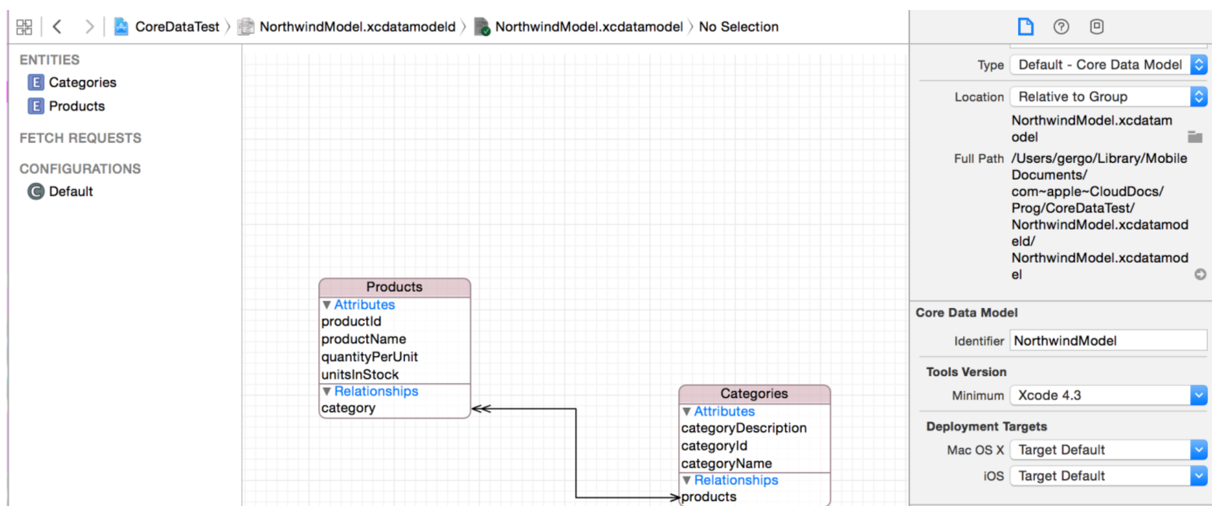


Figure 19: The CoreData Designer UI in XCode

On this UI the tables and the relationships can be built by drag and drop. On the left part you can also set additional properties to the model. The most important property is the Identifier. With this identifier you can pull out the model in code.

For relationships a "deletion rule" can be defined and depending on this rule the reference object also can be deleted when the referencing object is deleted. For every attribute a default value can be defined here and if a new item of the entity is created then the attributes are automatically set to this default value.

After the mapping is created on this UI Xcode is able to create Objective-C classes based on the defined entities by clicking on “Create NSManagedObject Subclass” under the “Editor" menu. In CoreData every database entity is a subclass of NSManagedObject. Although you can also work with the framework without generating strongly typed classes it is very uncommon to directly use NSManagedObjects since in that case you have to reference the column names with raw strings and that means that the code still compiles if you change the name of a property.

For the two classes defined above Xcode generated this code:

```
@class Categories;

@interface Products : NSManagedObject

@property (nonatomic, retain) NSString * productName;
@property (nonatomic, retain) NSString * quantityPerUnit;
@property (nonatomic, retain) NSNumber * unitsInStock;
@property (nonatomic, retain) NSNumber * productId;
@property (nonatomic, retain) Categories *category;

@end
```

[Products.h]

```
@class Products;

@interface Categories : NSManagedObject

@property (nonatomic, retain) NSNumber * categoryId;
@property (nonatomic, retain) NSString * categoryName;
@property (nonatomic, retain) NSString * categoryDescription;
@property (nonatomic, retain) NSSet *products;
@end

@interface Categories (CoreDataGeneratedAccessors)

- (void)addProductsObject:(Products *)value;
- (void)removeProductsObject:(Products *)value;
- (void)addProducts:(NSSet *)values;
- (void)removeProducts:(NSSet *)values;

@end
```

[categories.h]

The generator is a one path generator which means when a table is processed the generator knows only the tables at that point which already were processed before. So when one table references another one which was not yet processed then the type of that table is still NSManagedObject instead of the concrete type. In this case you have to start the generator again and when it next time processes the table it already knows the referenced table too so it chooses the right type for the reference.

The entry point to the database is an instance of the class named *NSManagedObjectContext*. As in Entity Framework the *DbContext* this class is a hook to go to create objects on the database, set attributes, and query it.

To insert a new object into the database you need to call the *insertNewObjectForEntityForNameinManagedObjectContext* method. The first parameter is the name of the entity and the second parameter is the *NSManagedObjectContext*.

```
NSManagedObjectContext* context = self.document.managedObjectContext;

Categories* category = [NSEntityDescription insertNewObjectForEntityForName:@"Categories"
      inManagedObjectContext:context];

category.categoryDescription = @"TestDescription";
category.categoryId = [NSNumber numberWithInt:1];
```

```
category.categoryName = @"TestName";
```

This method returns an *NSManagedObject** or a subclass of it if you generated strongly typed classes with Xcode. The properties on the newly created objects are either set to nil or to the default value if they are provided.

The *NSmanagedObjectContect* is normally provided by a subclass of *UIDocument* (most of the time it is a *UIManagedDocument*) which has auto saving functionalities and it takes care of saving the changes on the disk.

To query the database you can use the *NSFetchRequest* class. The first thing you need is an Entity. The result of a query is a set of entities from the same type. There is no way to get back from the database different type of entities (for example some Categories and some Products) with a single query. You pass this parameter when you construct an *NSFetchRequest* as a string to the *fetchRequestWithEntityName* static method of the *NSFetchRequest* class.

With the *fetchLimit* property you define how many items you want to get back and the *fetchBatchSize* property defines how many items the query returns at a time. When for example your query matches 1 million elements in the database CoreData does not load all 1 million elements, it just loads 1 million placeholders and fetches the items when you need them. The batch size defines how many of them are loaded in one fetch.

With the *sortDescriptors* property which is a type of *NSSortDescriptor* the result can be sorted. Since the return of the request is an NSArray this array can be sorted and this sorting is based on this *sortDescriptor*. There are multiple ways to create an instance of *NSSortDescriptor*. One of them is the static method on the class:

```
sortDescriptorWithKey:(NSString *)key
        ascending:(BOOL)ascending
        selector:(SEL)selector
```

Here *key* specifies on which columns the results will be sorted. With the selector property you specify which selector is used to decide the order. There are some predefined selectors for this for example the *localizedStandardCompare* selector. (Apple inc.)

The *sortDescriptors* property where you attach the sort descriptor to the *NSFetchRequest* instance is an array. The reason for this is that as with any other major database technology you can order the items based on multiple criteria (like for example first name and last name).

The most important property on *NSFetcRequest* is the predicate property of type *NSPredicate*. This says which of those entities you want, so in SQL terms it is basically the where term. *NSPredicate* basically has its own predicate language.

One way to create an *NSPredicate* is to use the static *predicateWithFormat* method on the *NSPredicate* class:

```
NSPredicate* predicate = [NSPredicate predicateWithFormat:@"categoryDescription contains %@",
        categoryDescription];
```

With NSPredicate one can define much more complex predicates. The whole description of NSPredicate can be found on the Apple Developer website (Apple inc.).

To summarize the discussion about *NSFetchRequest* here is an example how to query the Categories table.

```
NSFetchRequest* request = [NSFetchRequest fetchRequestWithEntityName:@"Categories"];
request.fetchBatchSize = 20;
request.fetchLimit = 100;
request.sortDescriptors = nil;

NSPredicate* predicate = [NSPredicate predicateWithFormat:@"categoryDescription contains %@",
        categoryDescription];

request.predicate = predicate;
NSError* error;
NSArray* result =  [context executeFetchRequest:request error:&error];
```

2.5 Summary

This section introduced some relevant, state of the art concepts and products in the field of database synchronization frameworks and database access frameworks on mobile devices. The solutions from Microsoft and Oracle and the proposal by (Tatjana Stankovic, 2010) introduced in (2.1 Platform Independent Database Replication Solution Applied to Medical Information System) offer a great solution for database synchronization and most of the concepts regarding change tracking and conflict resolution already solve the problem stated in the problem description. One aspect of these the products from Oracle and Microsoft is that they do not support SQLite and you cannot directly synchronize these databases with a mobile application without an additional layer of abstraction.

They also do not fulfill the "Minimal infrastructure" requirement defined in 1.3 Requirements: The Microsoft SQL server needs agents to be configured and running outside of the DBMS and this feature is not included in every version. Oracle GoldenGate is also additional software which needs to be installed and configured.

Entity framework and Core Data are designed for application developers to access databases from OOP languages and Core Data supports mobile applications on the iOS platforms. Both of them solve the problem of accessing relational data in object oriented code and CoreData also solves the problem of database access specifically in mobile application, but only for iOS and none of them support platform independent solution for the synchronization problem between a database running on a Smartphone and another database running on a classical database server.

The conclusion of this chapter is that for the synchronization process the two replication products introduced here fulfill every requirement regarding the synchronization itself, but they are neither completely platform independent nor they can be included directly in a mobile application. On the other hand CoreData and Entity Framework show how to integrate a database access layer with user code. Therefore the direction to solve the stated problem is to implement a similar API as CoreData and EF do and add extra functionality to them to not only support local database access, but also synchronization.

3 Proposed solution

When it comes to mobile devices we have currently 3 very important players on the market. These are the IOS operating system from Apple, Android from Google and Windows/Windows Phone from Microsoft. According to Gartner in 2014 Q3 on 83.1% of the sold Smartphones Android was installed, on 12.7% is running IOS and 3% of them were sold with Windows on it. (Gartner, 2014) This means that if the framework supports these three platforms then it already covers 98.8% of the Smartphone market based on the 2014 Q3 numbers.

On the Tablet-PC market we also see similar numbers: in 2013 Android had 61.9% market share, IOS had 36% and Windows had 2.1%. This means that by supporting these 3 platforms the framework will automatically support practically 100% of this market segment. (Gartner, 2014)

Of course there are many differences between the platforms and code sharing is still problematic across these difference systems. Fortunately the database typically lives in a non-UI related layer, which means the classical code sharing problems are much easier to solve. One other good thing is that all the platforms support SQLite, moreover on Android and on iOS it is by default installed and used by the operating system itself and on Windows it is supported by Microsoft and it can be installed via a nuget package. The second observation regarding code sharing is that all the platforms have their own programming language and programming model, but there is one language which can be compiled on every platform. This language is C++.

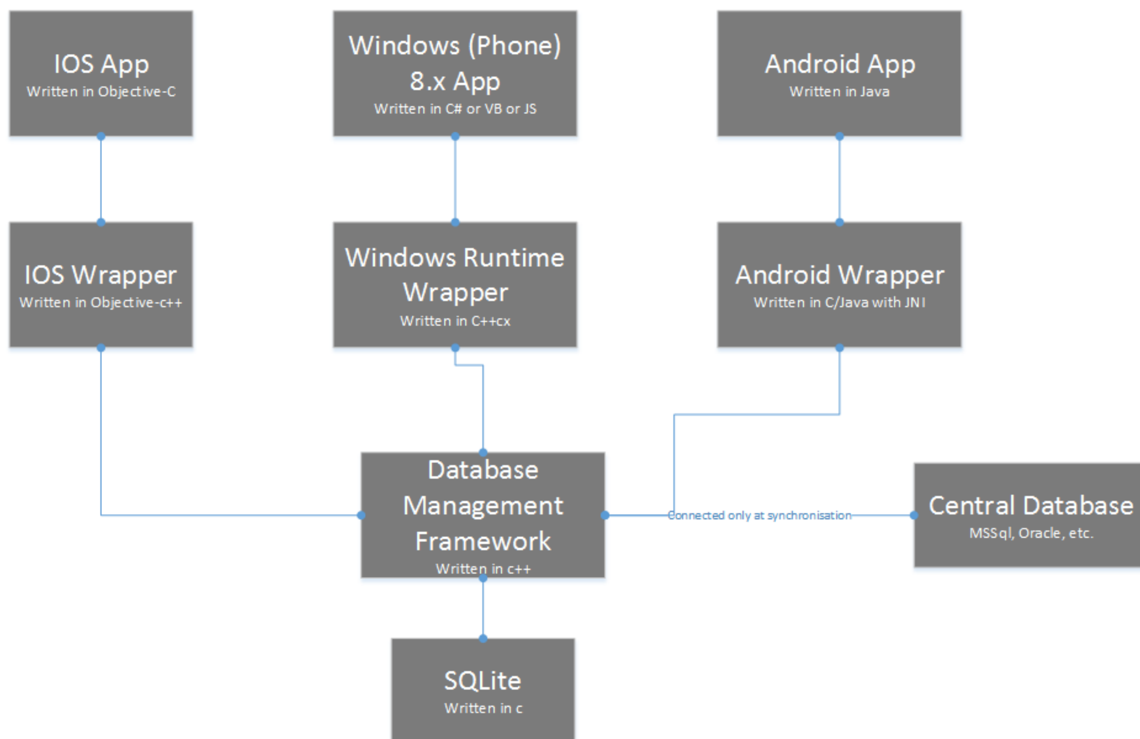


Figure 20: The proposed architecture

On Figure 20 you can see the high level architecture of the proposed solution: at the bottom there is the SQLite library and this layer manages the files on the file system in which the database is stored. On top of that layer is the heart of the proposed system: this is the database management framework called MobileSync which is responsible for the change tracking in the local database and for the synchronization between the local SQLite database and the remote (central) database. This layer is written in C++ which follows the C++ standard; therefore it can be compiled by any C++ compiler. Since this framework must be available on every mobile platform the strategy is that for every platform there will be a thin layer implemented to wrap the C++ code and project the classes from the C++ layer into C#, Java, Objective-C and to other languages. These wrappers are responsible for the type conversion between the different programming languages.

Of course not the whole database management system is projected to these higher level languages: only public classes are available in order to interact with the local database and start the synchronization process. Every other detail like change tracking is completely hidden from users.

Finally on top of these wrapper classes is the user code. These are normal mobile apps, with a local database and this local database can then be synchronized to the central database and this synchronization is triggered by user code.

3.1. SQLite

Based on (sqlite.org) and on (Allen, 2010).

SQLite is an open source, file based database engine designed by D. Richard Hipp written in C. Normally a classical database system runs as a separate process and clients can connect to this process via a network. SQLite does not follow this principle; it is always linked into an application like a normal library, so the client and the database run both in the same process. The whole database is contained in a single file which is by default not encrypted, but there is also support for encryption in the API. This file is platform independent and can be copied between different platforms regardless of endianness and other things. SQLite uses B-Tree representation for both tables (the raw data) and indices.

SQLite supports most of the SQL standard, but there are also some “not supported“ SQL features. For example SQLite does not support foreign keys. The reference clause will be parsed, but it won't be forced afterwards, so you can insert rows which reference non existing rows from another table, although a foreign key relationship is defined. There are also some alter table commands which are not supported and there is only a limited support for triggers.

3.2 The SQLite type system

SQLite has a very unique type system which is different from the classical static type system used by other database systems. In a classical DBMS the type of value is determined by its column. In contrast, in the SQLite type system "the data type of a value is associated with the value itself, not with its container”. So every SQL statement which works on a static type system also works on SQLite, but the SQLite type system allows more, so commands which were denied for example by a SQL Server because of a type system violation still run on SQLite without any problem, because the database engine can handle such commands.

To give a taste about this type system let’s see an example:

```
gergo — sqlite3 — 80×22
Gergos-MacBook-Pro:~ gergo$ sqlite3
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> CREATE TABLE TABLE1(COLUMN1 int);
sqlite> INSERT INTO TABLE1(COLUMN1) Values(1);
sqlite> INSERT INTO TABLE1(COLUMN1) Values(2);
sqlite> SELECT * FROM TABLE1;
1
2
sqlite> INSERT INTO TABLE1(COLUMN1) Values('3');
sqlite> SELECT * FROM TABLE1;
1
2
3
sqlite> INSERT INTO TABLE1(COLUMN1) Values('this is text');
sqlite> SELECT * FROM TABLE1;
1
2
3
this is text
sqlite> 
```

Figure 21: SQLite types

In the first line I created a table called Table1 with one column called Column1. The type of this column is defined to int. In the next two statements I insert 1 and 2 into this table and the next statement lists the tuples from the table. Until this point nothing special happens, this would run on every other DBMS, too.

After the first select statement the value '3' is inserted, but this time as a text and not as an int. Since the column is defined as int (in a SQLite term the type affinity of this field is int, but more on this later) the value is converted from text '3' into int 3. This is very unique to SQLite since other Database Management Systems would throw an error for this line.

The last insert statement inserts a text, but this time the value cannot be converted to int, so SQLite stores it as it is and as you see in the output of the last select statement the last tuple of the table really contains text.

So the type of a value is not determined by the column, but still every value has a type in this type system. The SQLite type system defines different storage classes and every type is a member of one of these classes.

- **NULL**. The value is a NULL value.
- **INTEGER**. The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- **REAL**. The value is a floating point value, stored as an 8-byte IEEE floating point number.
- **TEXT**. The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB**. The value is a blob of data, stored exactly as it was input.

(sqlite.org)

As the name suggests every store class contains more types. The values from the different types inside a storage class are stored on the disk differently, but as soon as the values from a storage class are read into the memory they are always converted into the most general type, so the term type and storage class are most of the time interchangeable. For example in C++ or C normally we use the int type to store integer numbers, which is on today's system either a 4 or an 8 byte integer, but if the value can be stored as a 1 byte integer it is stored that way, but this difference on the C/C++ layer is not visible, since in C/C++ we always see that the type of the value is int.

This type system is inspired by scripting languages and it is also called Manifest typing or Duck-Typing. The philosophy behind Duck-Typing is really simple: "if something looks like a duck, if it quacks like a duck, if it moves like a duck then it must be a duck".

Now the obvious question is: what has this duck-typing to do with the defined type of a column and how does this influence the chosen type for a value in the example above? The concept here is that every column has a so called type affinity. When we define a column as a column which stores numbers (in the example above we defined our column as an int) then we define the type affinity of that column, so if we insert a text which looks like an integer ('3') then according to Duck-typing it must be the integer, and since the type affinity of the column is number the value is converted into an integer. If the value cannot be converted (like in the last insert statement above) then the type remains the original type of the value, similarly to our case text. SQLite defines every column to one of these affinities:

- TEXT
- NUMERIC
- INTEGER
- REAL
- NONE

To determine the affinity of a column these rules are used (from (sqlite.org)):

1. If the declared type contains the string "INT" then it is assigned INTEGER affinity.
2. If the declared type of the column contains any of the strings "CHAR", "CLOB", or "TEXT" then that column has TEXT affinity. Notice that the type VARCHAR contains the string "CHAR" and is thus assigned TEXT affinity.
3. If the declared type for a column contains the string "BLOB" or if no type is specified then the column has affinity NONE.
4. If the declared type for a column contains any of the strings "REAL", "FLOAT", or "DOUB" then the column has REAL affinity.

5. Otherwise, the affinity is NUMERIC.

3.3. Calling C++ from a higher level language

As it was mentioned before the most suitable language to solve the described problem is C++. But we also have to consider that most of the mobile applications are not written in this language, but in another higher level, platform dependent language like Objective-C or Swift for IOS and Java for Android and in C# or JavaScript on the Windows platform. This section describes how it is possible to call C++ code from these languages. Three solutions are introduced for three different platforms and all of them use different approaches. For example the Objective-C and C++ interoperability is completely based on the compiler since it understands both languages while the Windows solution uses a so called language projection layer with metadata and multiple compilers are involved. One important thing I would like to point out is that here we talked about code sharing and not about binary sharing. It is not possible to share compiled code between platforms and reuse it from different programming languages since the compilers on the different platforms generate code which is not binary compatible. The implanted C++ code must be compiled for every platform with a compiler for a given platform.

This kind of code sharing is not uncommon in software products. In the cpp-con Microsoft presented how they share C and C++ code across Windows, Mac and Android in the Office product (Tony Antoun, 2014) and Dropbox also gave insights into their code sharing across IOS and Android (Alex Allain, 2014).

This section gives an introduction to the techniques regarding wrapping portable C++ code on different platforms, but it does not go into details of the implementation (except that some code examples came from the implemented wrappers). Here are only the concepts presented and later sections go into implementation details and show how these concepts are used in the implementation.

3.4 Calling C++ from Objective-C

The iOS compiler is able to compile C++ and it is possible to call C++ code from Objective-C. This technology is called Objective-C++.

The classical approach is that we embed the plain C++ files into the iOS project and write wrappers in Objective-C++ to do the data conversion. Some types work out of the box, like int and double, and some of them need manual conversion. One example for the latter situation is when we have NSString on the Objective-C side and std::string or char* on the C++ side. These conversions typically take place in the wrapper classes.

As an example let say we have the following C++ class with a getHelloWorld method:

CppClass.h:

```
#include <string>

class CppClass
{
public:
    std::string getHelloWorld(){
        return "cppString";
    }
};
```

The wrapper class for this is an Objective-C class which has a classical header file in a .h file. The significant difference to a traditional Objective-C class is that the implementation file has the .mm extension instead of .m. This tells the compiler that the implementation contains C++ code.

The header file of the wrapper class would look like this:

ObjectiveC.h

```
#import <Foundation/Foundation.h>
@interface ObjectiveCClass
-(NSString*)getHelloWorld;
@end
```

As you can see this is still plain Objective-C, so you can include this header into any Objective-C class and it also usable from Swing.

And the implementation would be this:

ObjectiveC.mm:

```
#import "ObjectiveC.h"

@implementation ObjectiveCClass

-(NSString*)getHelloWorld{
    CppClass cppClass;
    std::string cppString = cppClass.getHelloWorld();
    return [NSString stringWithUTF8String: (cppString.c_str())];
}
```

As we can see the header file does not contain any C++ related code, so we can embed it anywhere across the iOS application. All the callings to the C++ layer and the data conversion takes place in the .mm files.

3.5 Wrapping C++ code into Windows Runtime Components

The older versions of Windows used the so called Win32 APIs and application developers could build application on top of this API. Since win32 is very low level API Microsoft provided wrappers and frameworks around it to improve the productivity of the developers. The most important framework was the .NET framework and the C# programming language. This is a completely managed environment with a virtual machine called Common Language Runtime (CLR) which runs so called Intermediate Language (IL) and the CLR uses the Win32 API via COM calls. There are compilers for several languages which can generate intermediate code. The most used languages are C# and Visual Basic, but there is also a C++ dialect called C++/CLI which can be compiled into IL.

With Windows 8 Microsoft introduced a new low level API to replace Win32 and this is the so called Windows Runtime and this runtime is reused in the Windows Phone operating system. The main problem with Win32 was that it was not designed for small tablet and Smartphone devices and with that API it was very hard to build responsive touch application. The new Windows Runtime API was completely written in C++ with the mind that it must be fast and it must support all the new requirements of the Smartphone's and tablet-PCs. But Microsoft did not want to change all the existing frameworks and languages, so keeping the C# and VB support was also very important.

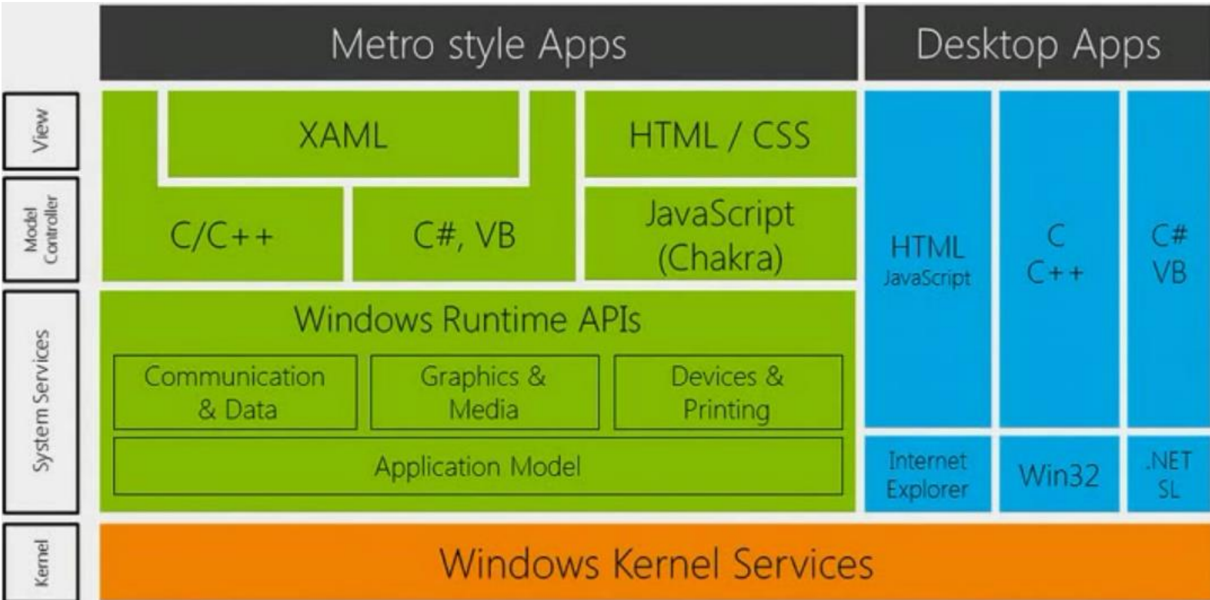


Figure 22: The windows infrastructure (Microsoft, 2012)

The supported languages by the new windows runtime API are C/C++, C#/ VB and JavaScript, so these are the languages today which can be used to build Windows tablet or Smartphone applications.

They way this works is that Microsoft developed a Metadata system with a Language Projection layer which basically projects every class from the Windows Runtime API into the language of the above layer. So there is only one C++ class in this API and it can be used from multiple languages.

And this is the point where we can talk about the implemented database synchronization framework and its integration to the Windows system.

It is possible to create Windows Runtime components which are basically C++ libraries and they run below the language projection layer. This means that the windows runtime component which wraps the existing C++ code is automatically projected into all the languages supported by the language projection layer. This way the code which is introduced in this document can be called from C#, VB and JavaScript on Windows and Windows Phone.

These Windows runtime components can be written in multiple languages, also in C# which is one of the most convenient ways to implement such a component. But the one problem we have here is that the original sync framework is written in C++ and it must be referenced from the Windows runtime component. In this case the only way to implement this is to create the windows runtime component also in C++, which is the bridge between the two layers and the foundation to the language projection layer.

Visual Studio already provides a project template for building Windows Runtime components in C++:

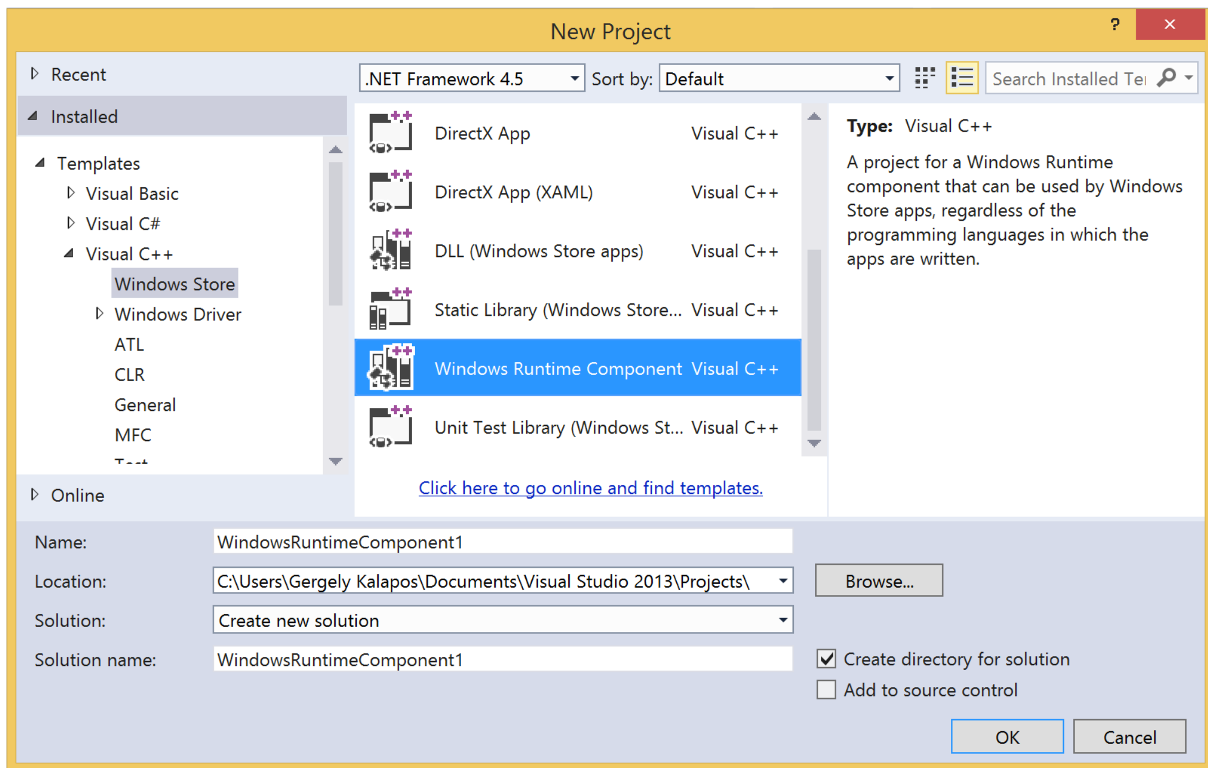


Figure 23: Creating a WinRT component in Visual Studio

One important thing about this project template is that the language is called C++, but the C++ which can be used in this project is not the standard C++. This dialect is called C++/CX and Microsoft created this C++ extension to enable programmers to write Windows Runtime components without directly using COM and any other complicated interoperability technologies.

The other option beside C++/CX to write these components would be to use the Windows Runtime C++ Template Library (WRL), which is a lower level library to create Windows Runtime components and it can be used instead of C++/CX. The C++/CX compiler automatically generates what one had to write in pure C++ syntax with WRL. The rest of this thesis ignores the WRL, because the implementation of MobileSync uses C++/CX. The reason for that is that everything that C++/CX has to offer is enough to solve the problems described in the requirements. For example instead of exceptions WRL uses HResults to report errors and one consequence of this would be that the Windows Runtime Component which wraps the shared C++ code would contain more error handling code without any benefit. (MSDN)

As it was mentioned before every Windows Runtime API is written in C++ and these are projected into the different higher level languages. This already implies that some parts of the classes written inside this layer are projected and obviously some of them aren't, because they are implementation details. This will be the same with the database synchronization framework. There will be classes which must be projected into other languages so users can use the framework, but the framework also contains many "private" classes which are used internally and they must be hidden from the users. Examples for this

are all the change tracking classes, since change tracking happens automatically and users are not allowed to control these change tracking classes.

3.5.1 Short Summary about C++/CX

The first important property of this C++ dialect is that everything defined in the C++ standard compiles with a C++/CX compiler too. Beside the standard C++ features there are some additional keywords in the language to support Windows Runtime components. The new keywords and symbols are borrowed from C++/CLI, which was introduced before and basically enables to write .NET code in C++. For readers familiar with this dialect I would like to point out that the C++/CX syntax is similar, but the meaning of the keywords is completely different. As I mentioned before Windows Runtime is not a managed environment, so for example there is no garbage collection at this level.

The first C++/CX keyword I would like to introduce is *ref*. It can stand before a class declaration and before the *new* keyword. If a class is marked with *ref* it tells the compiler that this class must be projected to higher level languages by the languages projection layer. So basically every C++ ref class with public visibility is a class which can be used from C# and JavaScript as any other class from the Windows Runtime framework. So for example the class which wraps the pure C++ class responsible for the management of the local database looks like this:

```
public ref class LocalDbManagerWrapper sealed
{
private:
        …
public:
        LocalDbManagerWrapper(Platform::String^ dbName);
        void CreateTable(WindowsDataTable^ Table);
        …
};
```

Another C++/CX specific syntax element is the ^ sign. It is used to define reference counted pointers which are projected into higher level languages. For example in this case the *PerformSelect* method returns a *WindowsDataTable^,* which is a reference counted pointer to an instance of the *WindowsDataTable* class which is also defined in the as a part of the Windows Runtime Component wrapper layer. Another non standard C++ keyword in the class is the *sealed* keyword after the class name. Every WinRT class which is projected by the language projection layer must be defined as sealed class and it prevents inheritance from that class (similar to the final keyword in the C++11 standard).

To create a class which is defined as a windows runtime component you also need to use non standard C++ syntax. For example the *PerformSelect* method creates a WindowsDataTable instance in C++/CX which is then passed to C# or JavaScript. The syntax in this case is this:

```
WindowsDataTable^ retTable = ref new WindowsDataTable();
```

Here *ref new* means that the created instance is reference counted and it can be accessed in C# and in JavaScript as any other object.

The other aspect of C++/CX beside the specific keywords like *ref*, *sealed* and the ^ sign is the set of available types: the projected classes are compatible with multiple other languages, so the classes defined in C++/CX must follow the rules of these other languages. To make this interoperability easier there are some predefined namespaces available with useful classes, for example the Platform or the Windows::Foundation::Collections namespaces.

An example for an interoperability issue is that in C# every class is a subclass of the Object class, but the C++ standard does not say that a class must have a parent class at all. To solve this problem the designers of C++/CX decided that every projected class is automatically subclass of the Platform::Object class which represents a generic parent class for projected classes and every projected instance can be converted into this type.

Another example is the different representation of strings: today every major programming language has a string class in its type system or at least some kind of concept to represent strings, but different languages solve this problem differently. C# uses the String class from the System namespace, and the C++ standard defines a string class in the standard template library (STL), which is also used by the shared C++ layer. There is a string class defined in the Platform namespace for C++/CX which is transformed to System.String by the language projection layer when in C#.

Many code base deals with collections and in fact collections are one of the most important components of the implemented framework: every table is a collection of rows and every row is a collection of fields. Here we also have a huge difference between the different layers: C# uses the List<T> class from the System.Collections.Generic namespace, and C++ uses std::vector<T>. C++/CX solves this with the IVector<T> class from the Windows::Foundation::Collections: every C# List<T> instance is projected into the IVector<T> class, and this IVector<T> then in the C++/CX layer can be converted into std::vector<T>. Here I would like to point out, that for example on IOS there is no need for this conversion step, since the NSArray can be directly converted into std::vector in an Objective-C++ class.

3.6 Calling C++ from Java with the NDK

This part is mainly based on (Gargenta, 2010)

A part of the Android framework is the Android Native Development Kit, or shortly NDK. The point of NDK is to embed native code (typically C or C++) into Android Apps. The

documentation explicitly says that using the NDK is not recommended for most of the apps, because it adds extra complexity to the code. One scenario to use the NDK is when performance is on the top of the requirement list and you can move the calculation intensive code into a C or C++ component. The second reason to use NDK is exactly the topic of this thesis: code sharing between Android and other operating systems.

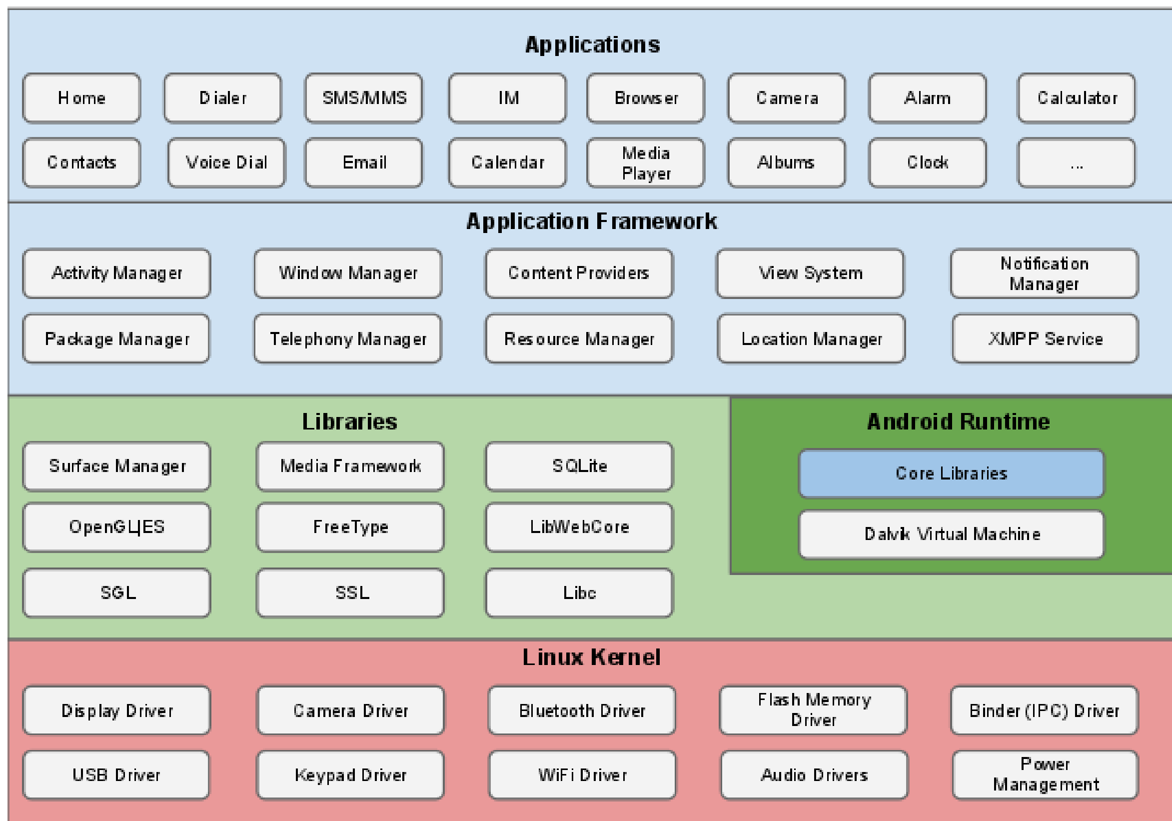


Figure 24: The Android infrastructure (Google inc.)

Similar to the Windows Platform Android is also built with a layered structure. On the bottom there is a Linux Kernel which provides the abstraction of the hardware. Writing code against the Kernel makes it possible that the same code can be used on different hardware.

On top of this layer there are Libraries. These are native libraries, typically written in C and in C++. As you can see one of the libraries is SQLite, so it is by default part of the Android system. The second piece of this layer is the Android Runtime including the Dalvik Virtual Machine. This is a VM which runs the code written by application developers. There are many differences between Dalvik and the JVM of oracle: The most important difference is that on Android every App has its own VM and the compilation is also different on the Android platform.

When an Android App is compiled, first from Java the classical Java Byte Code is generated. Additionally from this Java byte Code a so called Dalvik Byte Code is generated and this is what runs on the Dalvik VM.

On top of this layer there are the Application Frameworks and other Applications which use those frameworks. This is already a Java layer.

So the key takeaway of this short introduction of the Android Platform is that the bottom layer is written in native languages like C and C++ and on top of this layer App developers create Apps which run on the Dalvik VM. Beside that it is an important fact that with the NDK it is possible to build an application where one part of the App is written in Java and another part is written in a native language which uses the native libraries from the "Libraries" layer directly.

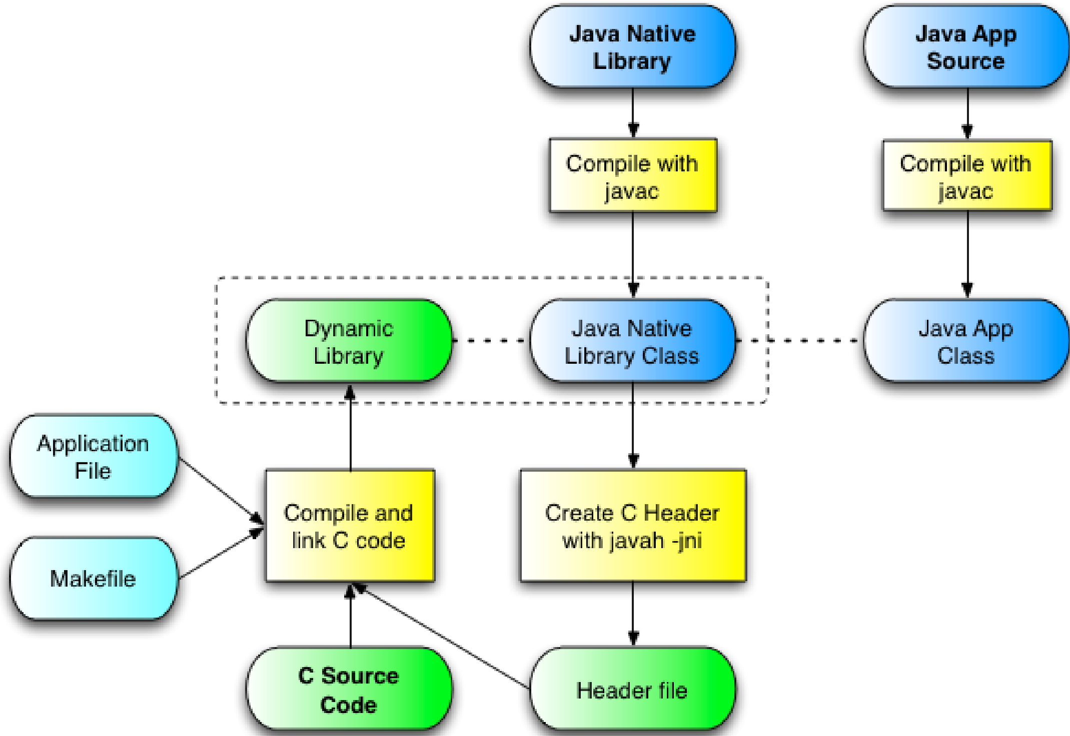


Figure 25: The Android NDK (Gargenta, 2010)

The way Java code calls into native code is defined by the Java Native Interface (JNI). This was already used before Android became popular and JNI itself can be used basically with every major JVM implementation.

In order to use native code in the implementation you obviously have two parts: the java classes which use C/C++ code and the C/C++ code which is basically the implementation of the code which is called by Java code.

An example for this is the *HelloJni* class

```
public class HelloJni extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        TextView  tv = new TextView(this);
        tv.setText( stringFromJNI() );
        setContentView(tv);
    }

    public native String  stringFromJNI();

    static {
        System.loadLibrary("hello-jni");
    }
}
```

In the static block the hello-jni is loaded. The name of the library is defined on the native side. The method which triggers the native code is the *stringFromJNI* method. The method is marked with *native*, which is a keyword in java and it tells the JVM that the implementation of that method is in a native library (which should be loaded by a System.loadLibrary call).

The corresponding C function is normal C code, which includes the jni header.

```
#include <string.h>
#include <jni.h>

jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env, jobject thiz )
{
       return (*env)->NewStringUTF(env, "Hello from JNI!");
}
```

The jni.h defines the types which can be passed from C to java like jstring and it also defines additional helpers like the JNIEnv.

These methods are the ones where all the type conversions take place between the C++ layer of the synchronization framework and the Java code.

All these native code is placed in an Android App project in the IDE under the jni folder.

There are two other files which are very important in order to wire up the pieces of an Android application developed with NDK.

The Android.mk file is a small GNU makefile fragment and this is where the modules are defined. The content of the Android.mk of the sample code is this:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := hello-jni
```

```
LOCAL_SRC_FILES := hello-jni.c
include $(BUILD_SHARED_LIBRARY)
```

This is where “hello-jni” is defined as the module name and this is exactly what is imported in the static block of the java code.

The second build configuration file is the Application.mk file. Here the architecture is defined agains which the build system builds.

```
APP_ABI := all
```

This is important, because different Android devices use different CPU architectures and since we compile native code here it won't run directly on a JVM, so it is important to build for the right CPU architecture.

3.7 Summary

The goal of sections 3.3-3.6 was to give a summery about the possible solutions to use pure C++ libraries on different mobile platforms. Of course there is a lot to tell about these technologies, but the core take away from this part is that all the 3 major mobile operating systems have some kind of support to project C++ libraries into higher level languages.

All of the 3 solutions presented in these sections are specified and supported by the vendors of these operating systems and the implemented framework uses these three solutions to project the parts of the framework which serve as the public API for App developers into higher level languages.

4. Details of the proposed solution for the synchronization problem

This section describes how it is possible to keep track of the changes in a relational database and then how the changes are synchronized between two databases in order to bring them to a state where both databases contain the same rows with the same data. This part is a general description of the framework and it gives a high level overview about the design and algorithms of the implemented synchronization system. The next section contains vendor specific descriptions like how to list all the tables in a database, or how to get the foreign keys between all the tables and also goes into details regarding the C++ implementation. For these kinds of low level problems it is impossible to give a general solution, so all such details can be found in later sections.

4.1 The initialization step

Let's say we have one table in the central server as the initial state and we would like to set up the database, so the system is ready to do synchronization at any time and clone the database schema to a client which will exchange data with this central database. In this description we consider a "one central database and one client" scenario, which automatically covers the "one server database and n clients" scenarios.

In this example the table for which we turn on the synchronization is called *Table1*, and it has an *id* column and some other columns (*Attribute1, Attribute2,…*)

Figure 26: Sync initialization step 1: Identify table to initialize

After identifying the table we add a new column for it. We call this new column RowGUID. The type of the column is Universally Unique Identifier (UUID). An UUID is a 16 byte number and it is standardized by the Open Software Foundation (OSF) and it is used in distributed systems to make sure that every component can generate a unique id independently from other components. With this column we can identify every row regardless of whether it has been already synchronized or not. For every row which is under change tracking there must

be a change tracking table created. In our case the name of the table is *Table1ChangeTracking*. The table has a *RowGUID* column which stores the RowGUID of the row for which a change tracking row is created. The Operation column identifies what kinds of changes were made on that table and the SyncRound stores the time of the changes as an integer.

The management of the change tracking table is described in the next section. Besides the new row in Table1 and the change tracking table we also create a new table called SyncDetails, where all the synchronization-related information is stored.

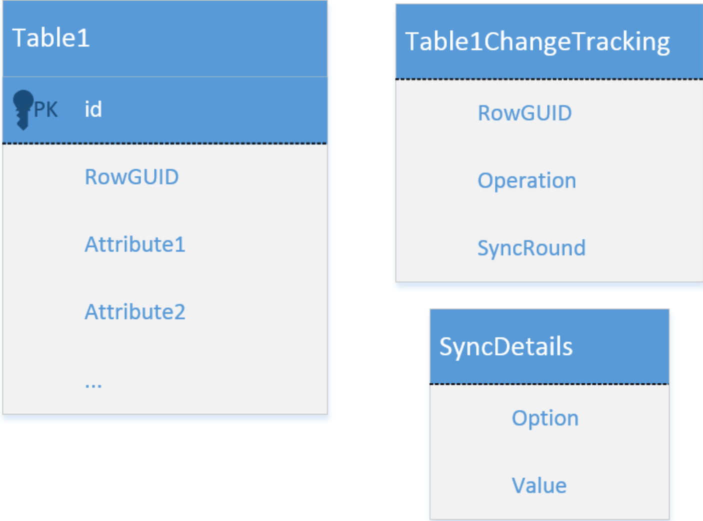


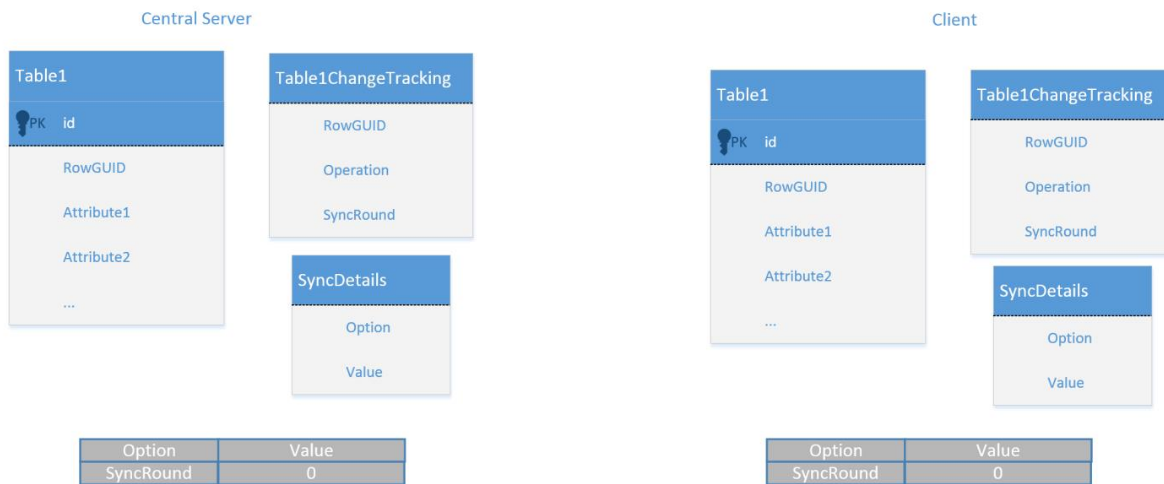
Figure 27: Sync initialisation step 2: The RowGUID column is added to the original table and the change tracking table was created, the new SyncDetails table is created as well

After the SyncDetails table is created a new row with Option "SyncRound" and Value "0" is inserted. This row measures the time in synchronization steps.

Option	Value
SyncRound	0

Figure 28: Content of the SyncDetails table in the initialization step

The next step is to copy the database schema to the client. In the initialization step the content of the SyncDetails is the same both on the server and on the client.



Option	Value
SyncRound	0

Option	Value
SyncRound	0

Figure 29: Sync initialization step 3: The server and the client contains the same schema

If Table1 contains any data then there is necessary to generate a RowGUID for every row. After that all the rows can be copied to the client.

4.2 Change tracking between two synchronizations

The goal of the change tracking tables is to keep track of the changes in a single database, so we know which rows were changed after the last synchronization. These rows will be examined during synchronization and processed according the synchronization rules, which will be introduced in the next section.

In a relational database 3 different things can be done with a row in a table: it can be inserted, modified or deleted. The different operations need different strategies to maintain the change tracking table. This section describes these change tracking strategies for the different operation. It is important to mention that the change tracking works differently on the central server and on the clients. The main difference is that on the client side every change tracking table is cleaned after the synchronization, since all the changes are processed in the local database. In contrast to this the server keeps track of every changes since different clients are at different stages, so nothing is removed from the change tracking table on the server side.

The next section describes the change tracking on the clients.

Insert

When a new row is inserted into a table the first thing what happens is that a RowGUID is generated for the new row and the row is inserted into the original table. The second step of the insert operation is that a new row is inserted into the SyncDetails. The *RowGUID* is the generated *RowGUID* of the new row, the *SyncRound* is the current *SyncRound* of the client and the *Operation* refers to Insert. Since at an insert operation we can be sure that the new row was never touched before, we can register the insert operation in the change tracking table without checking anything. (This is not the case at the update or deletes operations, as described later).

Table1

RowGUID	Id	Attribute1	Attribute2
3454b991-9393-4f11-9ad5-de38c8e4b3cf	1	value1	value2

Table1ChangeTracking

RowGUID	Operation	SyncRound
3454b991-9393-4f11-9ad5-de38c8e4b3cf	Insert	0

Figure 30: Change tracking for insertion

On Figure 30 a new row with id “1" was inserted. We can see the row in the original table and in the change tracking table. The two SQL statements which are sent to the database engine are the followings:

INSERT INTO Table1(RowGUID, Id, Attribute1, Attribute2) VALUES(‘3454b991-9393-4f11-9ad5-de38c8e4b3cf’, 1, ‘value1’, ‘value2');

INSERT INTO Table1ChangeTracking(‘owGUID, Operation, SyncRound) VALUES(‘3454b991-9393-4f11-9ad5-de38c8e4b3cf’, ‘Insert’, 0);

Delete

When a row is deleted the first thing the system does is that it stores the RowGUID of the deleted row in a temporary variable and then it removes the row from the table.

After this the change tracking table must be maintained. Here there are three possibilities:

a) If there is no row with the RowGUID of the deleted row in the change tracking table then the row which was deleted is already in the central database (either because the row was created on the same client where it was removed or because it was generated by another client and it was synced to the client which deletes it). In this case there will be a new row inserted into the change tracking table with the *RowGUID* of the deleted table.

   The purpose of the new row in the change tracking table is that at the synchronization time we know which row must be deleted from the original table on the server.

Table1

RowGUID	Id	Attribute1	Attribute2

Table1ChangeTracking

RowGUID	Operation	SyncRound
3454b991-9393-4f11-9ad5-de38c8e4b3cf	Delete	0

Figure 31: Change tracking after delete operation Case a

   The two SQL statements sent to the database engine are these:

   DELETE From Table1 Where RowGUID = '3454b991-9393-4f11-9ad5-de38c8e4b3cf'';

   Of course in most cases the where statement in the delete command does not contain the RowGuid, it is rather related to the other attributes.

   INSERT INTO Table1ChangeTracking(RowGUID, Operation, SyncRound) VALUES('3454b991-9393-4f11-9ad5-de38c8e4b3cf', 'Delete', 0);

b) If there is already a row in the change tracking table and this row is an insert statement we remove the row also from the change tracking table. The reason for this is that the row was created on the same client and it was not synced. This way we remove everything regarding the row, so it never makes it to the central database.

Table1

RowGUID	Id	Attribute1	Attribute2

Table1ChangeTracking

RowGUID	Operation	SyncRound

Figure 32 Change tracking after delete operation Case b

In this case the row with the given RowGUID is removed from both tables. The two SQL statements:

DELETE From Table1 Where RowGuid = '3454b991-9393-4f11-9ad5-de38c8e4b3cf'';
DELETE From Table1ChangeTracking Where RowGuid = '3454b991-9393-4f11-9ad5-de38c8e4b3cf'';

c) The last possibility is that the change tracking table already contains the RowGUID and the operation for that change is an Update. In this case we change the operation in this row from update to delete. With this at the synchronization we do not send the updated value to the server (at that point the updated value is removed anyway) but a delete command with the RowGUID. Table1 and Table1ChangeTracking are in this case in the same stage as in the 'a' situation.

Table1

RowGUID	Id	Attribute1	Attribute2

Table1ChangeTracking

RowGUID	Operation	SyncRound
3454b991-9393-4f11-9ad5-de38c8e4b3cf	Delete	0

Figure 33: Change tracking after delete operation Case c

The two SQL commands:

DELETE From Table1 Where RowGuid = ‘3454b991-9393-4f11-9ad5-de38c8e4b3cf’';

UPDATE Table1ChangeTracking SET Operation = ‘Delete' Where RowGuid = ‘3454b991-9393-4f11-9ad5-de38c8e4b3cf’';

Update:

When a row is updated there are also more cases which we have to consider.

a) If there is no row in the change tracking table with the same row we update the original row and insert a new row into the change tracking table with the RowGUID and with the *Update* operation

Table1

RowGUID	Id	Attribute1	Attribute2
3454b991-9393-4f11-9ad5-de38c8e4b3cf	1	updatedValue	value2

Table1ChangeTracking

RowGUID	Operation	SyncRound
3454b991-9393-4f11-9ad5-de38c8e4b3cf	Update	0

Figure 34 Change tracking after update, Case a

The two SQL statements:

Update Table1 Set Attribute1 = ‘updatedValue' where RowGUID = ‘3454b991-9393-4f11-9ad5-de38c8e4b3cf’;

Here of course there can be a different where clause.

INSERT INTO Table1ChangeTracking(RowGUID, Operation, SyncRound) VALUES(‘3454b991-9393-4f11-9ad5-de38c8e4b3cf’, ‘Update’, 0);

b) If the change tracking table already contains the RowGUID of the updated row and the operation for that track is an insert then the original row is updated and the change tracking table is not touched.

Table1

RowGUID	Id	Attribute1	Attribute2
3454b991-9393-4f11-9ad5-de38c8e4b3cf	1	updatedValue	value2

Table1ChangeTracking

RowGUID	Operation	SyncRound
3454b991-9393-4f11-9ad5-de38c8e4b3cf	Insert	0

Figure 35: Change tracking after update, Case b

This means that the row was created on the same client and it is not yet synchronized to the server. The effect of the insert operation row is that at the next sync the row is inserted into the server with the updated value.

In this case there is only 1 SQL command:

Update Table1 Set Attribute1 = 'updatedValue' where RowGUID = '3454b991-9393-4f11-9ad5-de38c8e4b3cf';

c) There is already a row in the change tracking table with the RowGUID and this row tracks an update operation.

Table1

RowGUID	Id	Attribute1	Attribute2
3454b991-9393-4f11-9ad5-de38c8e4b3cf	1	NEWupdated Value	value2

Table1ChangeTracking

RowGUID	Operation	SyncRound
3454b991-9393-4f11-9ad5-de38c8e4b3cf	Update	0

Figure 36: Change tracking after update, Case c

This means that the row was already at least once updated on the client, but it is not synchronized yet. In this case the system updates the row in the original table and leaves the update row as it is.

The SQL command:

Update Table1 Set Attribute1 = 'NEWupdatedValue' where RowGUID = '3454b991-9393-4f11-9ad5-de38c8e4b3cf';

4.3 Relationships between the tables

Until this point we only examined a database with one table. If we have more tables (and most of the time we have definitely more than one table) the tables can have relationships between each other. A typical relationship is a foreign key.

A foreign key in one table is a row which uniquely identifies another row in another table.

This foreign key relationship is one of the biggest challenges when we want to send data from one database to another one. Let's say we have Table1 with a column called FK1 which is a foreign key the K1 row of the Table2 table. In this case we have to make sure, that every time we access the FK1 column from Table1 then the corresponding K1 already exists.

When we create the tables at the initialization process we have to make sure that first we create Table2 with K1 and then we create Table1 with FK1. At the data synchronization step we also have to keep this order and first insert the data to Table2 and then to Table1. Otherwise we would want to insert a row into Table1 where we reference a row from Table2 via the FK1 row which does not exist at this point.

Of course this example is the easiest one. There can be more foreign keys into a database but it already shows the solution: we have to make sure that the order of the insert statements respects the foreign key relationships.

4.4 The Synchronization process

The hart of the framework is the synchronization process. The synchronization step is started by user code in a mobile app. In this step the delta of the data between the client and the server will be calculated and reduced to zero, so at the end of the synchronization we have the same data on the client and on the server side.

The main challenges at this step are the so called conflicts. A conflict can occur between a row in the remote database and a row in the local database. Two rows (one from the local, one from the remote database) are in conflict when they have the same RowGUID and this RowGUID is contained both in the Local and in the remote change tracking rows for a given synchronization. This means that if the time stamp of the last synchronization on the server is x, on the local database is y where y<=x then both in the local and in the remote change tracking table there is a tuple with this RowGUID where the times stamp is between x and y.

Now let's see how the synchronization works and how the conflicts can be solved. The synchronization itself can be divided into 4 stages for every table:

1. The first step is to get the ids of all the columns both from the local and remote side. For this the system reads the change tracking tables from both sides, so it knows which rows are involved in the current synchronization. This way we have two sets, the first one contains the involved tuples from the remote side the second one contains the tuples from the local database. The intersection of the two sets is the set of conflicts in the processed table. Every database synchronization framework provides some kind of automatic conflicts resolution logic and the framework implemented here is not an exception. The most important thing to keep in mind regarding conflict resolution is that it comes either with data loss or it decreases the data integrity.

Here are some concepts for automatic conflict resolution:

- **Server wins**: The precondition of this approach is that the data on the server is always "the truth" and if a client conflicts with this then the server is always right. For our framework this means that we always take the data from the remote side.

  So let's say R1 is a tuple representing the change tracking to a given row in the remote database and L1 is a change tracking table with the same id, but from the local database. With the "server wins" approach the conflict resolution function automatically returned the R1 tuple. Of course the change tracking tuple contains more information than the rowId, so the conflict can be for example, that the row is

updated in the local side and deleted on the remote side. All these facts are unimportant with this approach. One consequence of this is that clients that are synchronized more often will maintain the data in the central database with much higher intensity. The implemented framework follows this approach and currently this is the embedded conflict resolution algorithm implemented in MobileSync.

- Combine the two values: One solution which has negative effect on the data integrity would be to keep all the data from both sides. The formal description of this approach would be this: Let's say we have the CR tuple which is the change tracking row from the remote table and LR is the change tracking row with the same rowGUID from the local database. In the corresponding tuple TR on the remote side we have different values: TR{TR1, TR2, TR3… TRn} and we have the same structure in the local database in the corresponding tuple: TL{TL1, TL2, TL3…TLn}. In case of a conflict the conflict resolution function would create a new Tuple NT{TR1xTL1, TR2xTL2, TR3xTL3, TRnxTLn}, where x is an operation based on the database operation stored in CR and LR. For example if the original tuples contain text then the result tuple will contain the text both from the local and from the remote tuple if both change tracking row represent the update operation.

  The downside of this approach is that it is very hard to implement. The database system itself can contain the sum of all the values, but the applications built on top of the database also have to be able to handle such fields. One other problem is for example to deal with numbers. As (3.2 The SQLite type system) describes SQLite does not have classical static typing, so every column which has number type affinity still can stored values like " 1 | 4" which could mean that the value was updated both in the remote and the local database and on one side the new value is 1 and on the other side it is 4, so the system stores "1 | 4". The problem comes when we would like to store something like this for example in a Microsoft SQL Server database where the column has Int as its type. In this case additional work is needed to fully implement this approach

- Chose the winner based on values stored in the tuples: Another possible solution would be that the function examines the values in the rows and based on some rules it chooses one. These rules can be different, the '<', '>' could be used, or some kind of logic could be injected by the users.

2. After all the conflicts are resolved the next step is to process all the changes of the remote database and get the data into the local database. As it was introduced before, there is an additional *SyncDetails* table in the system which stored sync related information. One of the most important rows in this table is the timestamp of the last synchronization. This value is incremented every time on the server side after synchronization. So let's say at a certain point it is x on the central database (from the sync framework point of view in the remote database), then in the next moment a mobile device synchronizes to the central database. In

this case the timestamp becomes x+1 in the central database and the mobile device also updates its timestamp to x+1. Then the next device comes, which still has time stamp x in its SyncDetails table. In this case the framework selects every change tracking row with timestamp (as the syncround value) t, where $x<t<= x+1$. These are the rows that were changed between the last synchronization and the current one.

3. When the remote changes made it into the local database the local changes must be copied into the remote database. Since the local database is only changed by the application running on the same device the change tracking table is managed differently on the local database. Every change tracking table on the local database is cleared after the synchronization, so every line in a change tracking table is relevant for the current synchronization process, and this way there is no need to select or search for the correct change tracking tables. The system just takes the whole table and processes every row from it and then removes all the rows, so the change tracking table becomes empty.

4. The last step of the synchronization is a finalization and clean up step. The most important thing happening here is the synchronization of the timestamps. As it was already mention in the second step the timestamp of the remote database is incremented by one and this number is copied into the local *SyncDetail* table as the timestamp of the last synchronization. This way the local database knows until which time stamp it has all the data and at the next synchronization change tracking rows only with a time stamp bigger than this value are interesting.

5 Implementation details

This part continues the discussion started in Section 4 about the implemented framework, but it goes deeper into the vendor specific implementation details. Here I introduce some C++ features and techniques which are heavily used by the implementation. Beside C++ MS SQL specific solutions are also discussed.

5.1 The architecture of the implemented system

This part describes the architecture of the C++ code which implements all the concepts introduced in section 4. The chapter also highlights the most interesting design challenges and covers all the important architectural decisions. This is a bottom to top style description: first the very basic classes are introduced, which are on the bottom of the class hierarchy and do not have too much functionality. From there we move to classes which have more responsibilities and connect different parts of the system.

One of the most important basic classes is called DatabaseField. This class represents one field in the database and has 3 properties: the name of the column which identifies the value, the data type and the value itself.

```
struct DatabaseField{
  public:
     void* value_;
     DbType dbType_;
     std::string name_;
              …
}
```

One of the earliest design challenges was to decide how to store values in C++, since a database field can contain different types of data. The first idea was to define a template class. The problem with that approach was that templates in C++ are compile-time construct, so every time when a template is used in the code the actual type must be known when the code is compiled. This is unfortunately not the case when you implement a database synchronization framework, since you don't know in advance which types are used in which particular table. The second approach was to use the most general pointer, namely *void*.* This type was chosen to solve this problem combined with an enumeration. Every value is placed on the heap and every time the value is accessed the void* is casted to the correct type based on the value of dbType_.

One database row is represented by the DataRow class.

```
 class DataRow{
public:
   std::vector<DatabaseField> items_;

     …
};
```

The fields inside a row are stored in a vector.

One table is represented by the DataTable class:

```
      class DataTable{
public:
   std::string name_;
   std::vector<DataRow> rows_;
   …
 };
```

This class follows the same idea as the *DataRow* class. Here the rows are stored in a vector and beside that the class also has a name property which stores the name of the table.

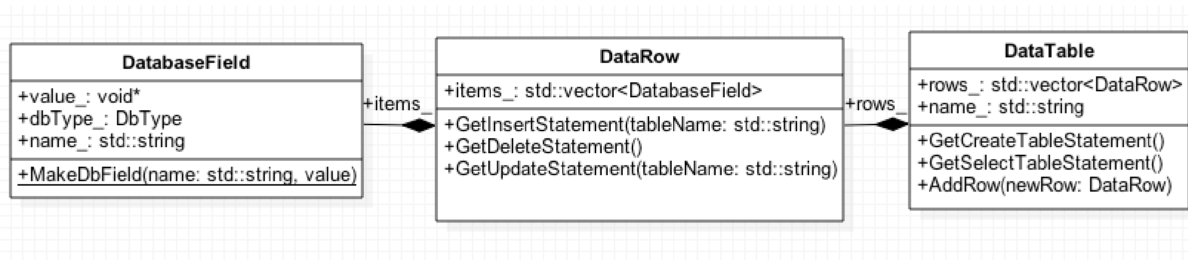


Figure 37: Tbe DatabaseField, Datarow and the DataTable classes

The change tracking tables and the change tracking rows are also normal database items, which could be represented by the *DataTable* and *DataRow* classes introduced above, but for convenience and to use static typing there are separate classes for them:

```
 class ChangeTrackingRow{
        public:
   std::string rowGuid_;
   int operation;
   int syncRound;


   static const std::string ROWGUIDCOLUMNNAME;
   static const std::string OPERATIONCOLUMNNAME;
   static const std::string SYNCROUNDCOLUMNNAME;

   std::string GetInsertStatement(std::string TableName);
   std::string GetDeleteStatementForId(std::string TableName);
};


      class ChangeTrackingTable{
        public:
```

```
    static const std::string CHANGETRACKINGNAMEPOSTFIX;

    std::vector<ChangeTrackingRow> rows_;
    std::string tableName_;

    static std::string GetCreateTableStatement(std::string tableName, DbSystem dbSystem);
};
```

ChangeTrackingRow contains three fields: the rowGUID of the referenced tuple from the original table, the operation type (because we have to know if we inserted, changed or deleted the referenced row) and the syncRound so we know at which point the row was modified.

The ChangeTrackingTable similar to a DataTable contains rows, but in this case the rows are instances of the class ChangeTrackingRow.

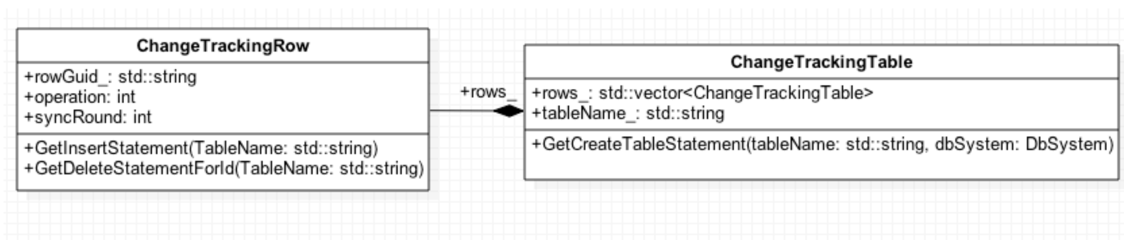


Figure 38: The ChangeTrackingRow and ChangeTrackingTable classes

The class *IDbConnector* is an abstract class and it defines the interface to communicate with a database. The "I" prefix is borrowed from the C# naming conventions, which shows that we deal with an interface here and not with a concrete class. In order to support a database system a specific *DbConnector* must implement all the functions defined by this interface.

```
class IDbConnector{
protected:
    std::string databaseAddress_;
    std::string databaseName_;

public:
    IDbConnector(std::string localFolderPath, std::string databaseName): databaseAddress_(localFolderPath),
databaseName_(databaseName){};
    virtual void OpenDb() = 0;
    virtual void CloseDb() = 0;
    virtual void PerformCreateTableStatement(std::string sqlStatement) = 0;
    virtual void PerformSqlStatement(std::string sqlStatemnt) = 0; //used for insert, update
    virtual DataTable PerformSqlSelectStatement(std::string sqlStatement, std::string tableName) = 0;

    virtual std::vector<ForignKeyRelation> GetForignKeys() = 0;
};
```

I think all the function names are self descriptive: the *OpenDb* function opens and the *CloseDb* function closes a database connection, and for every database operation there is a function to implement. The *PerformSqlStatement* gets a SQL statement in string

representation and forwards it to the database. Since there is no meaningful return value of this function it is normally used for delete and update operations.

Currently there are two implementations of this interface: *SqliteConnector* which implements the communication with the embedded, local SQLite database and the *MsSqlConnector* which implements the communication to a Microsoft SQL database server. The *MsSqlConnector* class uses the FreeTDS C library which is an open source implementation of the Tabular Data Stream (TDS) protocol. This protocol defines how data can be transmitted from a Microsoft SQL server to a client application.

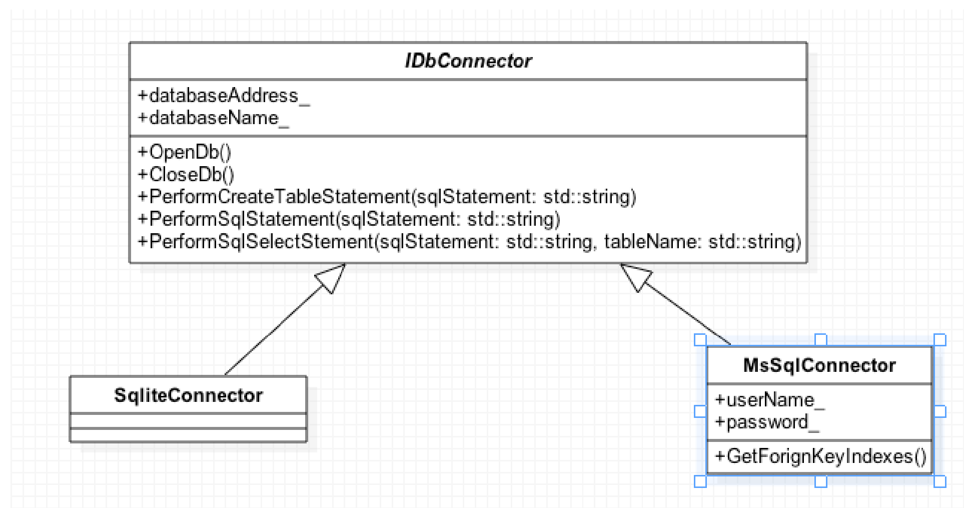


Figure 39 The IDbConnector and its implementations

At this point one can notice that in the design of the framework there is some symmetry: There is always a class which manages the local database and one which is responsible to the remote side.

The *SqliteConnector* class manages the communication between the framework and the local database. But the main class which is responsible for all the generated SQL statements is the *LocalDbManager* class. This class has an *IDbConnector* reference, which by default points to a SqliteConnector instance.

```
class LocalDbManager{

private:
    IDbConnector* localDbConnector_;
    ChangeTrackingManager changeTrackingManager_;
    DbSystem dbSystem = DbSystem::Sqlite;

public:
    LocalDbManager(IDbConnector* connector): localDbConnector_(connector),
changeTrackingManager_(localDbConnector_) {};

    void OpenDb();
    void CloseDb();
    void CreateTable(DataTable table);
    void PerformInsert(DataRow rowToInsert, std::string tableName);
```

```
    void PerformRawInsert(std::vector<DataRow> rowToInsert, std::string
    DataTable PerformFullTableSelect(std::string tableName);
    void PerformClearTable(std::string tableName);
    void PerformDeleteRow(std::string tableName, std::string rowGuid);
    void PerformUpdateRow(DataRow updatedRow, std::string tableName);
    std::vector<std::string> GetAllLocalTableNames();
  };
}
```

This class has all the functionality which is needed to manage the local database. This class implements all the logic which is described in Part 4. For example the *CreateTable* function not only creates a single table, but it also extends is with the *rowGUID* field and also creates the corresponding Change tracking table. Similarly to the *Perform* function also maintains the corresponding change tracking rows. Regarding change tracking one important design decision was made at the point where this class was created: the LocalDbManager class is not directly responsible for change tracking. As you can see in the interface the class has a private field called *changeTrackingManager_* which is an instance of the *ChangeTrackingManager* class. The *LocalDbManager* delegates all the tasks which have to do with change tracking to this class via method calls. The reason behind this decision was the "separation of concerns": One class should be responsible only for one task, so the *LocalDbmanager* is responsible for creating SQL commands to manage the local database and the *ChangeTrackingManager* is responsible for creating SQL commands for change tracking.

The interface of the *ChangeTrackingManager* is relatively simple:

```
  class ChangeTrackingManager{
  private:
    IDbConnector* localDbConnector_;

  public:
    static const std::string SYNCDETAILSTABLENAME;

    ChangeTrackingManager(IDbConnector* connector):
      localDbConnector_(connector){};

    void CreateTrackingTableLocal(std::string originalTableName);

    std::string GetCreateTrackingTableStatement(std::string
      originalTableName, DbSystem dbSystem);
    void CreateAndInitSyncDetailsTable();
    std::string GetCreateAndInitSyncDetailsTableStatement();
    void AddInsertInfo(DataRow newRow, std::string tableName);
    int GetSyncRound();
    DataTable GetItemsForGuids(std::vector<std::string> guidList, std::string tableName);
    void SetSyncRound(int newValue);
    void AddDeleteInfo(std::string rowGuid, std::string tableName);
    void AddUpdateInfo(std::string rowGuid, std::string tableName);
    ChangeTrackingRow GetChangeTrackingRowWithId(std::string rowGuid,
      std::string changeTrackingTableName);
  };
```

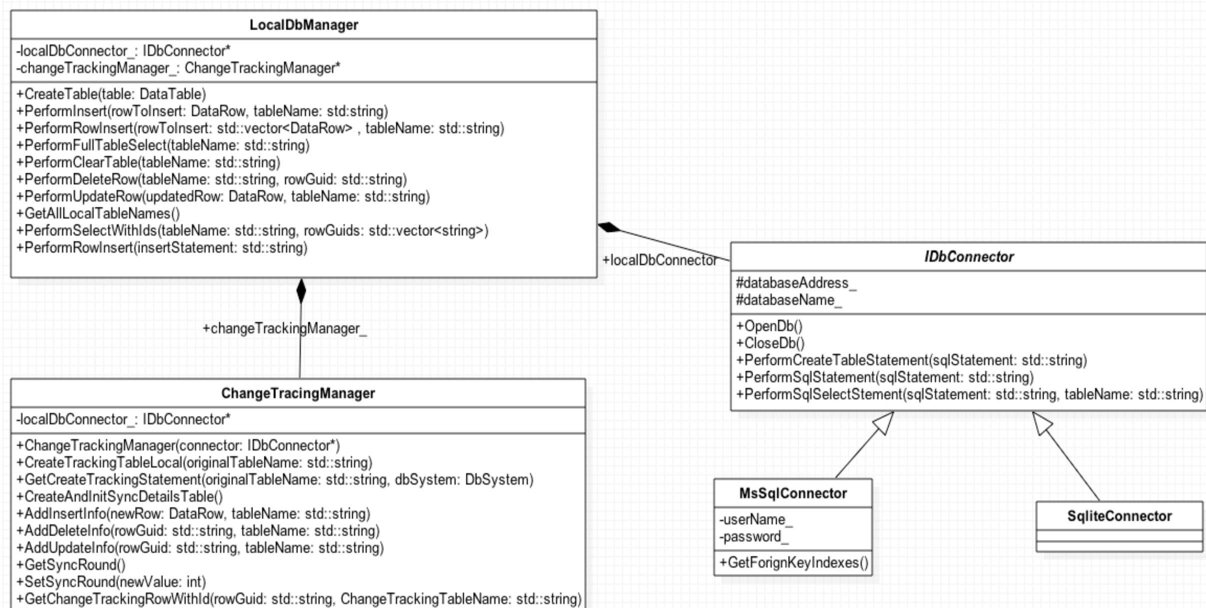


Figure 40 LocalDbManager and ChangeTrackingManager

For the "other side" the *RemoteDbManager* class is responsible. This class also has an *IDbConnector* reference, but this one points to the remote database which in the current implementation is always an *MsSqlConnector* instance. This class is used at every synchronization process and also at the initial synchronization, so you can see functions here like *GetAllRemoteTableNames* which returns a vector of all the table names from the target database and *SortTables* which sorts the tables based on the foreign keys and makes sure that the tables are processed in the right order, so every sql statement respect the foreign key relations. Beside these functions there it has the same functions as in the *LocalDbManger* to select, insert, update and remove rows.

```
class RemoteDbManager{
  private:
    IDbConnector* remoteDbConnector_;
    DbSystem dbSystem_;

    bool MatchForignKeyWithTables(std::vector<ForignKeyRelation>& forignKeys,    std::vector<DataTable>&
      tables);

    int GetIndexOfTable(std::vector<DataTable>& tables, std::string tableName);
    int GetIndexOfTableName(std::vector<std::string> tableNames, std::string tableName);

  public:
    int GetRemoteSyncRound();
    void OpenDb();

    RemoteDbManager(IDbConnector* dbConnector, DbSystem dbSystem):
      remoteDbConnector_(dbConnector), dbSystem_(dbSystem){};

    DataTable PerformSelect(std::string selectStatement, std::string  tableName);
    void PerformRowInsert(std::string insertStatement);
    void IncrementSyncRound();
    void PerformSqlStatement(std::string sqlStatement);

    std::vector<ForignKeyRelation> GetForignKeys();
```

```
    std::vector<DataTable> OrderTables(std::vector<ForignKeyRelation>&    forignKeys,
      std::vector<DataTable>& tables);
    std::vector<std::string> SortTables(std::vector<ForignKeyRelation>& forignKeys, std::vector<std::string>&
      tableNames);
    void UpdateTableWithRowGUID(std::string tableName);
    std::vector<std::string> GetAllRemoteTableNames();
  };
```

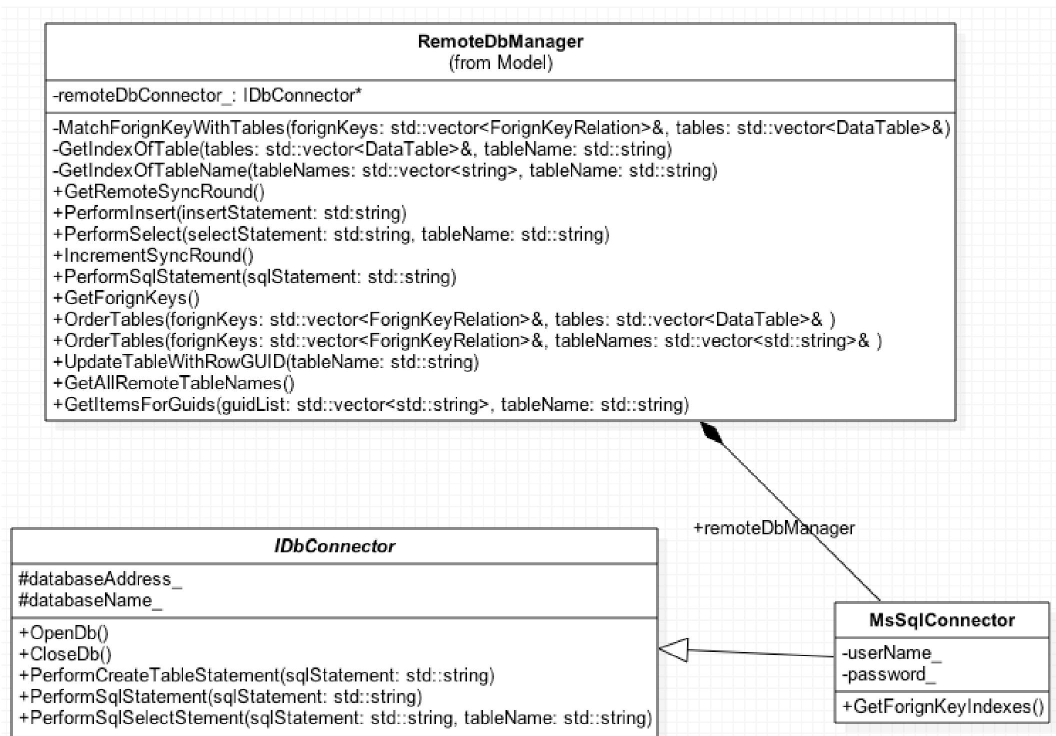


Figure 41 The RemoteDbManager class

And finally we arrived to the class which connects all the classes which were introduced before and this one is the *SyncManager* class. It has two public functions: *InitDb* and *Sync*. Both of them serve as facade classes to hide complex processes.

The *InitDb* function is triggered when the Smartphone or the Tablet-PC does not have a local database yet. It explores the remote database, creates all the tables locally with the change tracking tables and also creates the additional *SyncManger* table which store sync related information as discussed in Section 4. If it is necessary it also modifies the remote database: it can create change tracking tables on the remote side and it also adds the *rowGuid* column to every table if necessary.

```
class SyncManager{
   private:
   RemoteDbManager* remoteDbManager_;

   ChangeTrackingManager* changeTrackingmanager_;
```

```
LocalDbManager* localDbManager_;
InitialSyncManager* initialSyncManager_;

ChangeTrackingTable ConvertTableToChangeTrackingTable(DataTable table);
void ProcessRemoteChanges(std::string tableName, ChangeTrackingTable remoteChangeTrackingTable);
void ProcessLocalChanges(std::string tableName, ChangeTrackingTable localChangeTrackingTable);
void FinalizeTableSync(std::string syncedTableList);
void SyncTable(std::string tableName);
ChangeTrackingTable UpdateRemoteChangeTrackingTable(ChangeTrackingTable table);
ChangeTrackingTable GetRemoteChangeTrackingRws(std::string tableName);
void ProcessConflicts(std::string tableName, ChangeTrackingTable& localChanges, ChangeTrackingTable&
    remoteChanges);
void SolveConflict(std::string nameOfTheSyncedTable, std::string nameOfTheSyncTable, ChangeTrackingRow
    localRow, ChangeTrackingRow remoteRow);

MSLogger* _logger;

public:
  SyncManager(std::string serverAddress, std::string remotedatabaseName, std::string userName, std::string password,
    std::string localDbFolder, std::string localDbName);

  void Sync();

  void InitDb();
  InitialSyncManager* GetInitialSyncManager();

};
```

The Sync function starts the synchronization process. It uses both the LocalDbManager and the RemoteDbManager instances. This method basically first reads the list of the tables from the database and it triggers the private methods of the class with the correct parameters.

The constructor of the class has 6 parameters: the *serverAddress* is the address of the remote database server, the *remotedatabaseName* is the name of the database on the remote server (since in one SQL Server database instance more databases can be hosted) the *username* and the *password* fields store user credentials, the *localDbFolder* is the path of the folder which stores the SQLite database file on the file system of the Smartphone or Tablet-PC and the last parameter (*localDbName*) is the name of the database file.

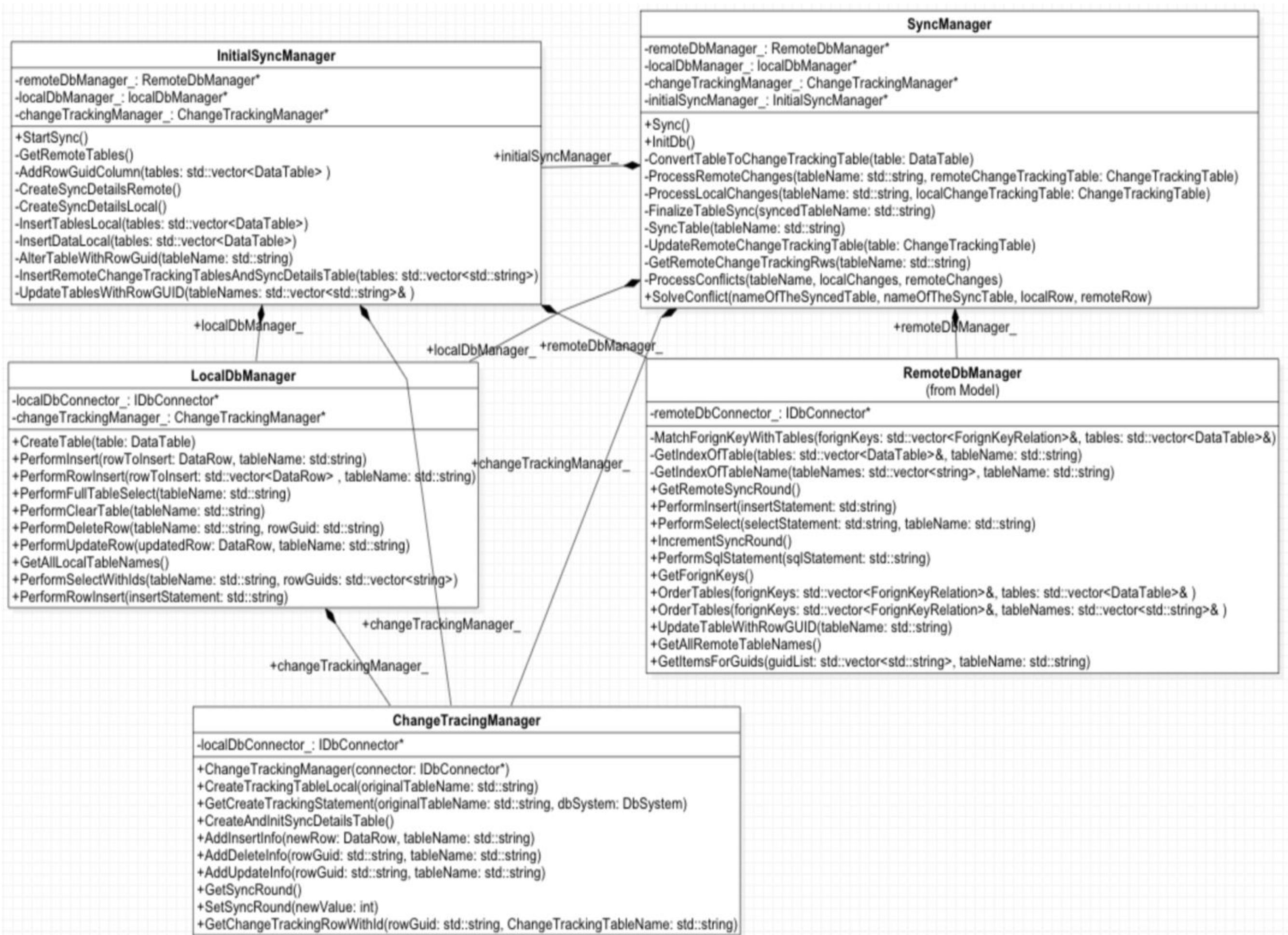


Figure 42 The SyncManager class

Another important design decision was to split the sync and the database initialization functionality into two classes. The *InitDb* and the *Sync* methods are different by their nature: the *InitDb* creates the local database and extends tables, the *Sync* method moves data and resolves conflicts. To simplify the interface there is only one class to manage both tasks, but the *InitDb* function is just a wrapper around the *InitialSyncManager* class and it delegates all the work to this class. This way the *SyncManager* in practice does only sync related tasks and for the initialization it uses the *InitialSyncManager* class.

Here is the interface of the InitialSyncManager class:

```
class InitialSyncManager
 {
 private:
    RemoteDbManager* remoteDbManager_;
    ChangeTrackingManager* changeTrackingmanager_;
    LocalDbManager* localDbManager_;
    std::vector<DataTable> GetRemoteTables();
    std::vector<std::string> AddRowGuidColumn(std::vector<DataTable> tables);
    void CreateSyncDetailsRemote();
    void CreateSyncDetailsLocal();
    std::vector<DataTable> InsertTablesLocal(std::vector<DataTable> tables);

    void InsertDataLocal(std::vector<DataTable> tables);
    void AlterTableWithRowGuid(std::string tableName);
```

```
    void InsertRemoteChangeTrackingTablesAndSyncDetailsTable
                    (std::vector<std::string> tables);
    void UpdateTablesWithRowGUID(std::vector<std::string>& tableNames);

  public:
    InitialSyncManager(RemoteDbManager* remoteDb, ChangeTrackingManager*
changeTrackingManager, LocalDbManager* localDbManager):
remoteDbManager_(remoteDb), changeTrackingmanager_(changeTrackingManager),
       localDbManager_(localDbManager){}
    void StartSync();
  };
```

The UML diagram of the whole system shows the most important classes and the connections between them: On top of the system there is the *SyncManager* class which delegates the initialization work to the *InitialSyncManager* class. In order to access the local and the remote database they have a reference to a *RemoteDbManager* and to a *LocalDbManager* instance (in fact they share the same instance). These two classes use the two *IDConnector* implementations which are the *MSSqlConnector* and the *SqlteConnector* classes and these classes operate on instances of the *DataTable* class which was introduces at the beginning of this section.

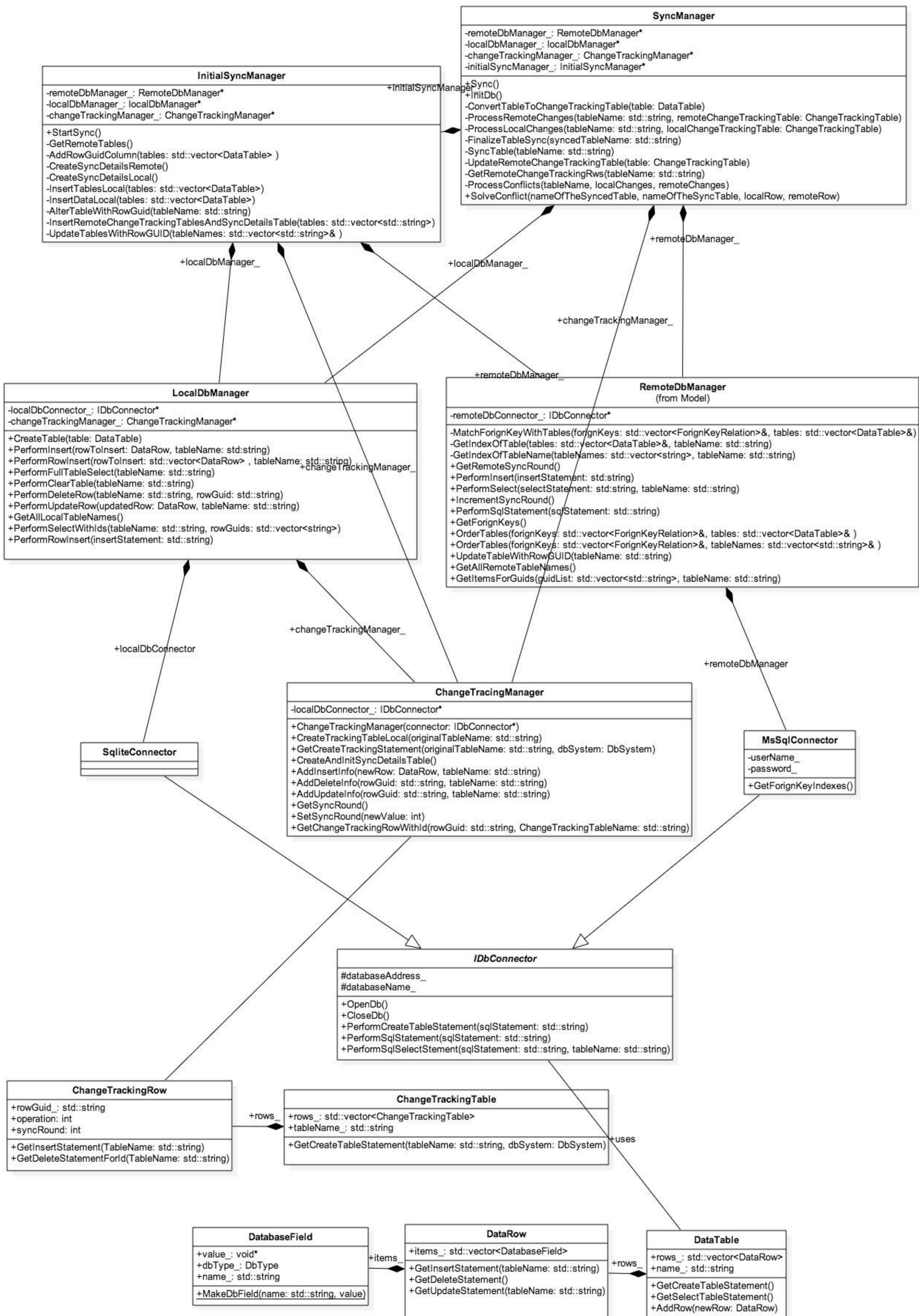


Figure 43 The UML Diagram of the MobileSync C++ layer

5.2 C++11 move semantic in the DataTable and DataRow classes

In general we can say if a framework is fast then it also uses less CPU and/or memory resources. In the case of a framework which runs on mobile devices it is obvious that the performance is on the top of the requirement list, since efficiency means in this case longer battery time. The framework is built with the new C++ 11 standard. One goal of the new C++ standard was to make it easier to write fast and low resource consuming code.

The most used C++11 features by the implemented framework is the so called move semantic. This section explains what move semantic is and how is this feature used by the implemented framework and what advantages it gives.

As the "The architecture of the implemented system" section described the most important class of the framework is the *DataTable* class which contains a vector of all the rows inside a table in a *DataRow* class and this class contains all the fields from a row, where the fields are represented by the *DatabaseField* class.

So both *DataTable* and *DataRow* are typical containers: *DataTable* contains rows and *DataRow* contains *DatabaseFields*. The two database manager classes (*LocalDbManager* and *RemoteDbManager*) have functions which return *DataTabeles*. One example is the *DataTable PerformFullTableSelect(std::string tableName)* function. By looking at this function without knowing anything about the C++11 standard one can immediately say that this function has a terrible performance.

Typically in the body of such a method an instance of the container is created (in this case a *DataTable*) which stores the item. This is represented in Figure 44:

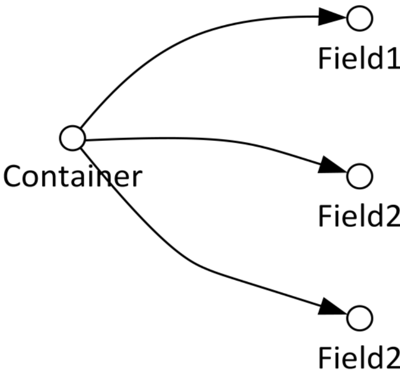


Figure 44: C++ container

So a container is created on the stack which stores its items (in this case rows and field) on the heap.

Now at some point we want to return this container from the function and the return value is used by the caller function, so we have a line somewhere in the code like this:

```
DataTable table = localDbManager.PerformFullTableSelect(“table1”);
```

What happens at this point is that the copy operator of the temporary container created in the *localDbManager.PerformFullTableSelect* function and all the items are copied into the table variable.

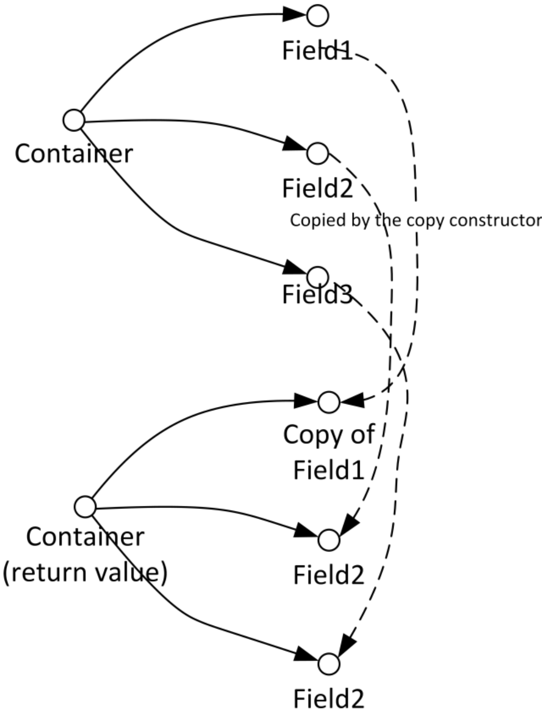


Figure 45 Copy constructor in collections

The bad thing happens right after the copying: the execution hits the end of the function and the destructor of the *DataTable* when the function returns is called and removes the original items:

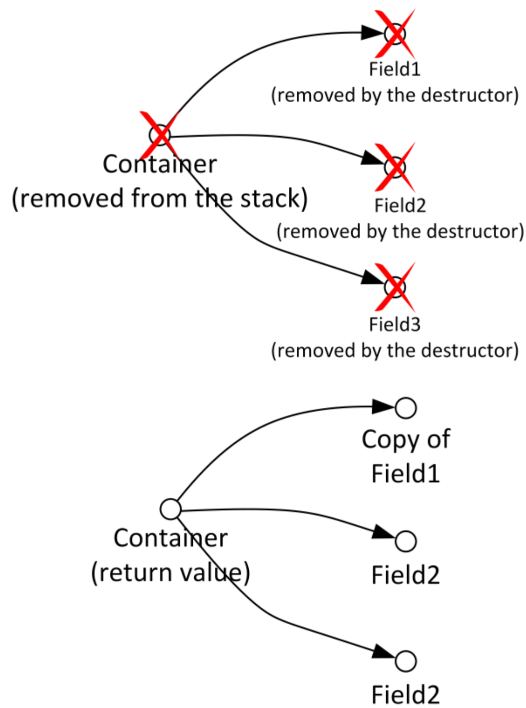


Figure 46 Returning a container without move semantics in C++

So what happens here is that we create a temporary container which stores its item on the free store, then the user code of this function wants to use these values, so there is another container there and we copy the items from the temporary container into the other one and remove the original items immediately. The temporary container and all of its elements die after a very short time.

To avoid this very expensive and unnecessary step the classical solution was to return a pointer of the container which is created inside the function, so to have a high performing function with the older C++ standard one would expect a signature like this:

```
DataTable* PerformFullTableSelect(std::string tableName).
```

And this would introduce the classical dilemma: Who is responsible to destroy this *DataTable* which is on the free store? And this question cannot be answered universally, so the best way is to have the original signature and still avoid the copying.

The C++ standard committee addressed this issue in the C++11 standard and introduced the concept of move semantics.

The idea is that instead of copying the items from the temporary container we can also "steal" them.

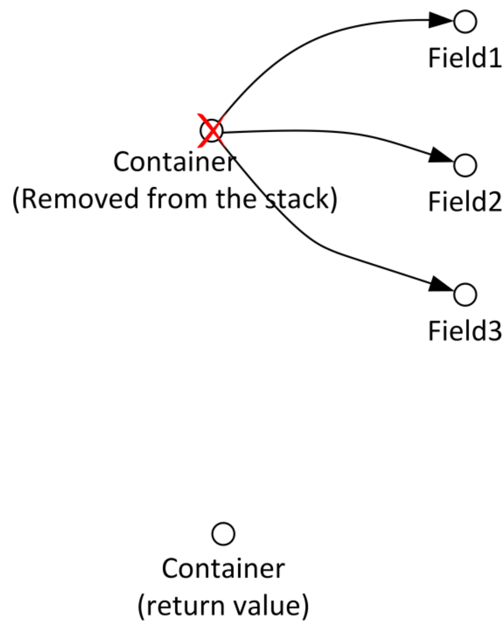


Figure 47: C++ move semantics step one: doe not destroy references from the temporary object

The initial step is similar to the one before (Figure 46 and Figure 46), but instead of copying all the items the new container can also get the addresses of the items and instead of copying them it is also possible just to store the old addresses and set the addresses in the old container to null.

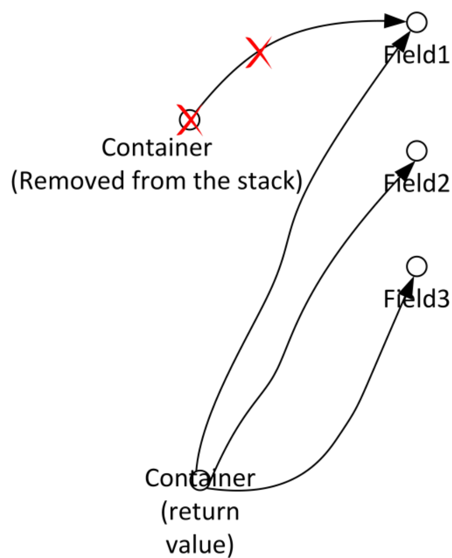


Figure 48: C++ move semantics step two: steal the pointers from the temporary container

So here the temporary container is also destroyed, but before its destructor would remove the items on the free store we take all the addresses of the items, the new container stores

them and sets the pointers in the original container to null, so the destructors does not destroy the items on the free store:

This way we reuse the items which are created on the free store by the temporary container and we still return the container itself and not a pointer.

This is the basic idea behind move semantics, which gives a very convenient way to implement the sync framework by a very efficient and still readable code.

The next section explains how the C++ layer of MobileSync uses this feature to increase both performance and code maintainability.

5.2.1 Rvalues and rvalue references and the constructors of the DatabaseField class

The valueness historically describes that a term is either on the left or on the right side of an expression. So for example in the expression int i = 0; i is an lvalue and 0 is an rvalue. The original definitions from the C language of rvalue and lvalue is this: "An *lvalue* is an expression e that may appear on the left or on the right hand side of an assignment, whereas an *rvalue* is an expression that can only appear on the right hand side of an assignment." (Thomas Becker, 2013) With the user defined types in C++ this definition is not enough because if we have an instance of our class then it can be both on the left and on the right side of an operator (e.g.: Myclass a; a = a;)

Therefore a newer definition was created: "An lvalue is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator. An rvalue is an expression that is not an lvalue" (Thomas Becker, 2013).

With this definition the difference between lvalue and rvalue is cleared and now we can move on with the term "rvalue reference" towards move semantic.

Following the notation of lvalue references which are noted by the "&" sign the C++ standard defines the "&&" sign to identify rvalue references. So if MyClass is a type then an rvalue reference to MyClass is MyClass&&. Rvalue references are expressions which are rvalues, but you can still on some way get a reference for the expression.

It is also possible to pass rvalue references to functions:

```
processMyClass(MyClass&& instance);
```

This behaves like a function overload, so beside the MyClass&& version you can still have an lvalue reference and an instance version:

```
processMyClass(MyClass& instance);
```

processMyClass(MyClass& instance);

The compiler always decides which version of the function is called based on the normal overload resolution rules.

Although you can overload any function by adding an rvalue reference version we typically use this with constructors and the assign operator:

```
DatabaseField(DatabaseField&& other): dbType_(other.dbType_),  name_(other.name_), value_(other.value_){
            other.value_ = nullptr;
    }

    DatabaseField& operator=(DatabaseField&& other){
        name_ = other.name_;
        dbType_ = other.dbType_;
        value_ = other.value_;

        other.value_ = nullptr;

        return *this;
    }
```

As one can see the *DatabaseField* class has a constructor and an assignment operator which takes an rvalue reference. This is important, because the STL containers take advantage of the rvalue references, so for example if an rvalue reference is passed into the *push_back* function it follows the expected behavior which is described above.

Now what the copy constructor makes is that it takes the address if the *value_* field from the old item, stores this value (the address) in the *value_* field of the new item and after this it sets the value of the *value_* field of the old one to *null*. So basically the new item "steals" the *value_* field, and this way the destructor of the old item cannot remove the *value_* on the free store and we avoid an unnecessary copy operation and increase the performance of the code.

Now let's examine the function which returns the *DataTable*:

```
DataTable SqliteConnector::PerformSqlSelectStatement(std::string sqlStatement, std::string tableName){

    sqlite3_stmt* ppstm;
    sqlite3_prepare_v2(db, sqlStatement.c_str(), -1, &ppstm, NULL);
    int cNumber = sqlite3_column_count(ppstm);

    std::vector<std::string> types;

    //get the decltypes for the columns
    for (int i =0; i<cNumber;i++) {
        types.push_back(sqlite3_column_decltype(ppstm, i));
    }


    std::vector<DataRow> rows;

    while(SQLITE_ROW == sqlite3_step(ppstm)){
        std::vector<DatabaseField> row;
        row.reserve(3);
```

```
        for (int i=0; i<cNumber; i++) {
            std::string name = sqlite3_column_name(ppstm, i);
            std::string dbType = types[i];
                ...
            row.push_back(DatabaseField::MakeDbField(dbType, value));
        }

        rows.push_back(std::move(row));
    }

    return DataTable(tableName, std::move(rows));
}
```

The first interesting line is the one where a *DatabaseField* instance is pushed into a vector:

```
row.push_back(DatabaseField::MakeDbField(dbType, value));
```

What happens here is that the MakeDbField returns an rvalue reference and since the vector class have an overload of the push_back method for rvalue references there is no second instance created here, the *value_* field which stores the value of the database field is created only once on the free store, which is a huge save in both CPU- and memory resources, since this can be a very long text or a big binary data.

The next interesting line is where one row is inserted into the vector which stores all the lines:

```
rows.push_back(std::move(row));
```

Here the *std::move* function is used. As I mentioned before the *push_back* function has multiple overloads:

```
void push_back (const value_type& val);
void push_back (value_type&& val);
```

The first one was there already in the C++98 standard, the second one came with C++11.

Now since we know that the line which we want to insert into the vector is a temporary variable we want to hit the second function. But what we have inside the for loop is a normal value type, which at the overload resolution matches the first function (meaning it would be copied and destroyed causing performance decrease). To solve this very common problem the Standard Template Library provides the *std::move* function which makes an lvalue to an rvalue reference. The name of this function is a little bit misleading, because it does not move anything. In this case the push_back function of the *std::vector* implements the move semantic and it moves out the values from the temporary variable into the vector items. The *std::move* function is just a helper, so the overload resolution at compile time chooses the right *push_back* implementation.

The last line where move semantic is used again is the return statement:

```
return DataTable(tableName, std::move(rows));
```

When we look into the implementation of the DataTable class we see that it also has a contractor with move semantic:

```
DataTable(std::string name, std::vector<DataRow>&& rows): name_(name), rows_(std::move(rows)){};
```

This forces the vector class to use its constructor for rvalue references, so there is nothing very special here. The reason to use the *std::move* function in the return statement above is the same as with the *push_back* function before: we want to hit the right method, in this case not the normal copy constructor, but the one with move semantics.

So as we have seen in this section that, by using move semantic and rvalue references it is possible to avoid returning pointers from a function but still have the same performance. This increases code maintainability and readability, since by using move semantic the caller code of the function returns a *DataTable* value type and not a pointer, so the code does not have to destroy objects on the free store and there is no chance to create memory leeks by using the function.

5.3 Initial sync step with MS SQL Server

As it is described in Section 4 the first step to use the synchronization framework is to set up the local and the remote database. The framework is designed for systems where a central database is already in use and the mobile application must be connected for this system.

When the mobile application is installed on a Smartphone or on a Tablet-PC the local database on that device is empty. The initialization step reads the tables and the schema of these tables on the remote database, extends it with the RowGUID and creates the *SyncDetails* table to keep track of all the synchronization related data. After this step the whole database structure is copied to the local database including already stored data.

To read the list of tables SQL Server provides *the sys.Tables* table.

With the

```
select * from sys.Tables;
```

command we get the list of all the tables in the database.

The next step is to get the schema of the individual tables. For this the INFORMATION_SCHEMA.COLUMNS can be used. For example to get the structure of the table called *Table1* the SQL command is this:

```
SELECT COLUMN_NAME, DATA_TYPE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = N'Person'
```

6. Testing

This part describes how the implemented framework was tested and what kinds of automated methods were used to make sure that one change during the development does not break some other functionality. Software testing is of course a very broad topic and there is a lot to say about it. This part only describes the aspects of software testing which were relevant to this project.

In the earlier days of software engineering companies sometimes confused the concept of software testing with manually trying out the software. Fortunately today software testing and test automation is a matured topic and there are lots of great frameworks which support test automation. In general we can say that software testing has nothing to do with trying out the software. It is much about writing code that makes sure that in every point of the software life cycle all the functionalities work correctly and the software fulfils the requirements.

MobileSync is tested on two levels: there are unit tests and integration tests.

The motivation behind the integration tests is that all the components must work together. On top of the sync framework there is an application which generates tables, tuples and stores them into the database and most of the time this application also wants to retrieve the data. In the middle there is the implemented synchronization framework which generates SQL statements based on the actions of the application. On the bottom of the system there is the SQLite database and an MSSql database on the server side and these systems must be able to process the sql statements generated by the sync framework. Integration tests make sure that all these components work together without caring about the intermediate results. The goal with integration tests is to make sure that all the pieces in the system work together correctly.

Unit tests are employed to test one component of the system at a time independently from the others. In this project a sync framework was implemented, so what the unit tests cover is the framework itself without any database or mobile application.

6.1 Integration Testing

To implement the tests the XCTest framework from Apple was used in the Xcode IDE. Xcode is the standard Integrated Development Environment (IDE) on the Mac OS and it can be used to develop Mac, IOS and other applications, including C++ applications. By default Xcode uses the LLVM compiler architecture so any LLVM language can be used in the IDE very easily. This is relevant to this thesis since both C++ (the sync framework is written in this language) and Objective-C (the sample IOS application is written in this language) are

supported by the LLVM architecture. The advantage of the XCTest framework is that it is out of box integrated with Xcode, so no special component needs to be installed.

XCTest is very often used to implement unit tests, but this does not mean that it cannot be used for integration testing. In this case one test case doesn't test a specific component; it rather tests one feature in an end-to-end manner.

Every test class must inherit from the *XCTestCase* class and every method containing the world “test" at the beginning of the method name represent one test. Of course one test class can have multiple test methods.

Inside the test methods as with any other testing framework we can use assertions to check if the system returns the expected values. There are multiple predefined assertions, for example *XCTAssertEqual*, *XCTAssertGreaterThan*, *XCTAssertNil*, and the *XCTAssert* itself.

One other functionality provided by the *XCTestCase* are the setUp and tearDown methods. The setUp method is automatically called before every test method; the tearDown method is called after the test methods.

To have an example the test class which tests the local database manager class is introduced here.

The class itself is called *MobileSyncIo_LocalDbOperations_IntegrationTests* and has 4 methods*: testInsert, testDelete, testSelect, testUpdate*. The methods test the corresponding database operations and check if everything went well. Since this is an integration test there is no faked or mocked component included, so every data goes to the SQLite database and is retrieved from it.

The tests are written in Objective-C.

The interface of the class only defines two helper functions: one for initializing the sample table used by the test and one for inserting a row:

```
@interface MobileSyncIo_LocalDbOperations_IntegrationTests : XCTestCase
-(void)initPersonTable;
-(void)insertRow;
@end
```

As it was mentioned before the *setUp* method runs before the test methods, here we remove the database file and recreate the whole database. This way we make sure that when we insert a table there is only one row in the table, which makes testing more convenient. Of course this is only true for this test.

```
- (void)setUp {
    [super setUp];

    [DatabaseTestHelper ClearDbFiles];
    localDbManager1 = [[LocalDbManagerWrapper alloc] initWithDbName:LocalDb1Name];
    [self initPersonTable];
}
```

The tearDown method only closes the database:

```
- (void)tearDown {
   [super tearDown];

   [self.localDbManager1 CloseDb];
}
```

The *testInsert* method tests if one row is inserted into the table and we call the *PerformSelect* method then the table really can be read from the database:

```
- (void)testInsert {

   [self insertRow];

   IosDataTable* selectedTable = [self.localDbManager1 PerformSelect:PersonTableName];
   XCTAssertEqual(selectedTable.Rows.count, 1, @"The Person table must containt 1 row");
}
```

The *testDelete* method works similarly, except that here 0 row is expected in the table:

```
- (void)testDelete {

   [self insertRow];

   IosDataTable* table = [self.localDbManager1 PerformSelect:PersonTableName];

   [self.localDbManager1 PerformDelete:table andOnRow:0];

   IosDataTable* retTable = [self.localDbManager1 PerformSelect:PersonTableName];

   XCTAssert([retTable.Name isEqualToString:PersonTableName ]);
   XCTAssertEqual(retTable.Rows.count, 0, @"Test removes the last row, we exprect 0 row");
}
```

6.2 Sample application

In order to test the implemented framework a sample application was built on top of the Northwind database.

This section introduces this sample application. The goal here is to show user code which is built on top of MobileSync.

Northwind is a sample database form Microsoft and it is mainly used for learning purposes for Microsoft SQL Server. The project can be downloaded from https://northwinddatabase.codeplex.com. It contains examples for the most important database concepts like foreign keys, stored procedures, views, so it is ideal to test a synchronization framework.

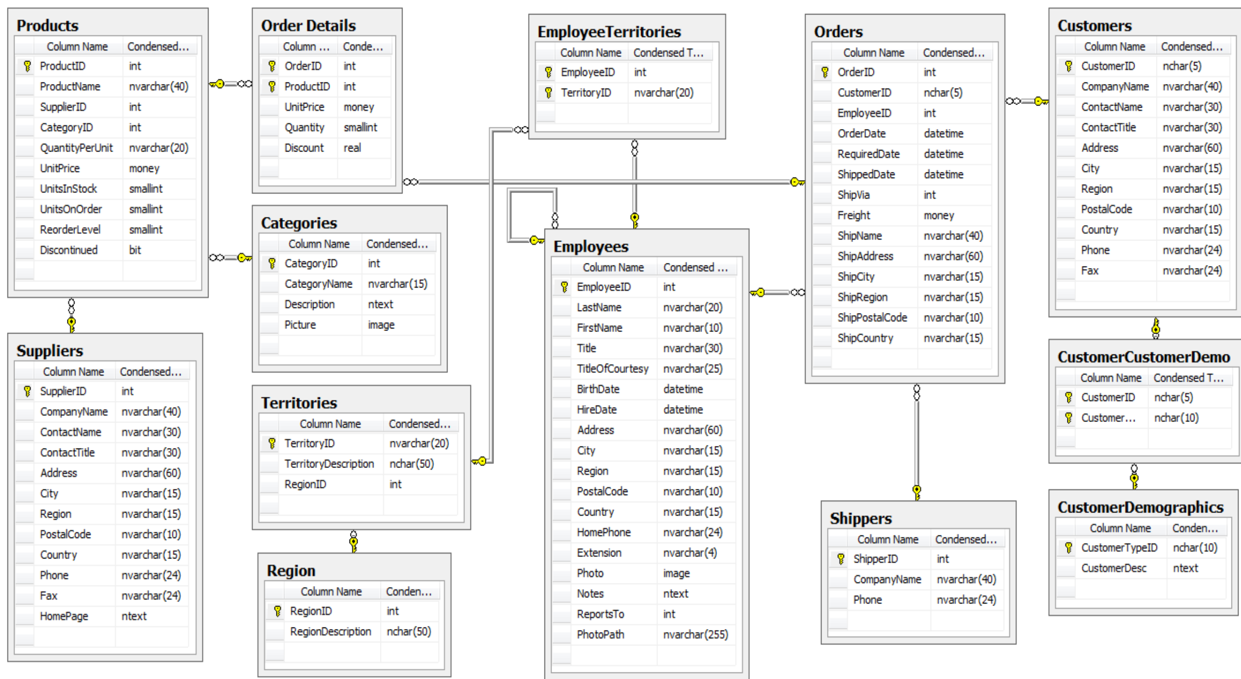


Figure 49: The Northwind Database (Microsoft)

On top of this database an iPhone and a Windows Store application were built and they synchronize their data via the MobileSync framework.

The sample application enables users to see their employee data and to change it. On the login screen the two input fields expect the first and the last name of an employee. If this employee exists then by clicking the “Login” button users can manage the data in the database for the selected employee.

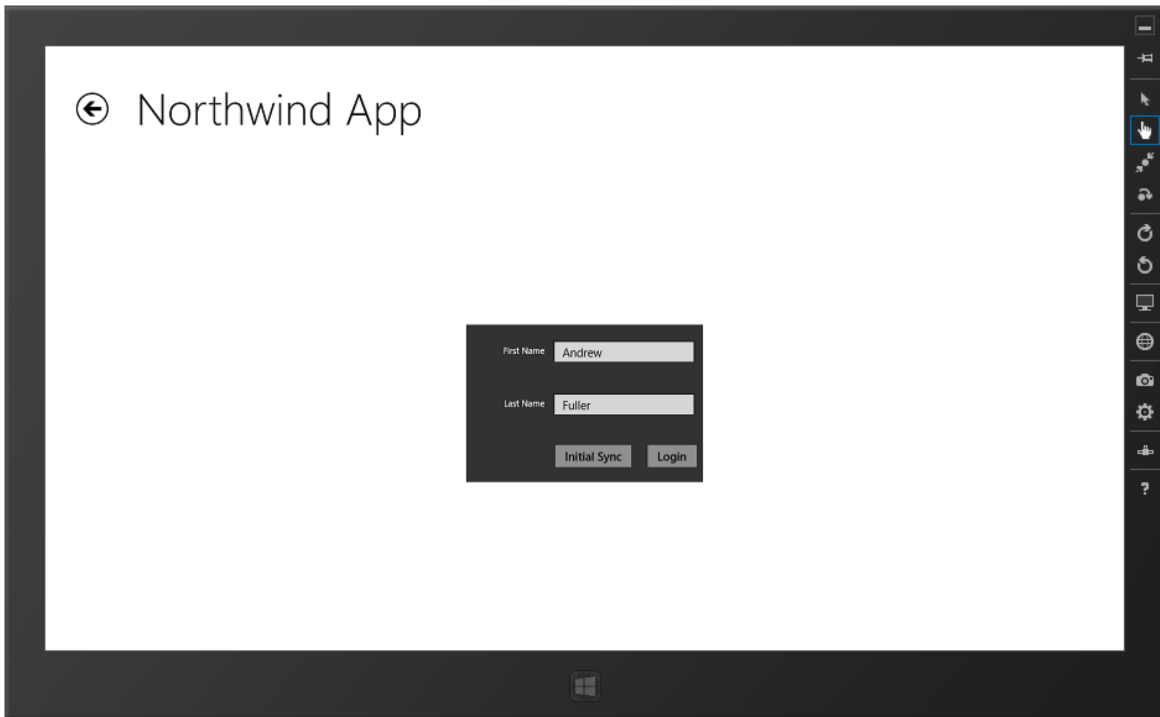


Figure 50:The login page of the sample application on a Windows Tablet

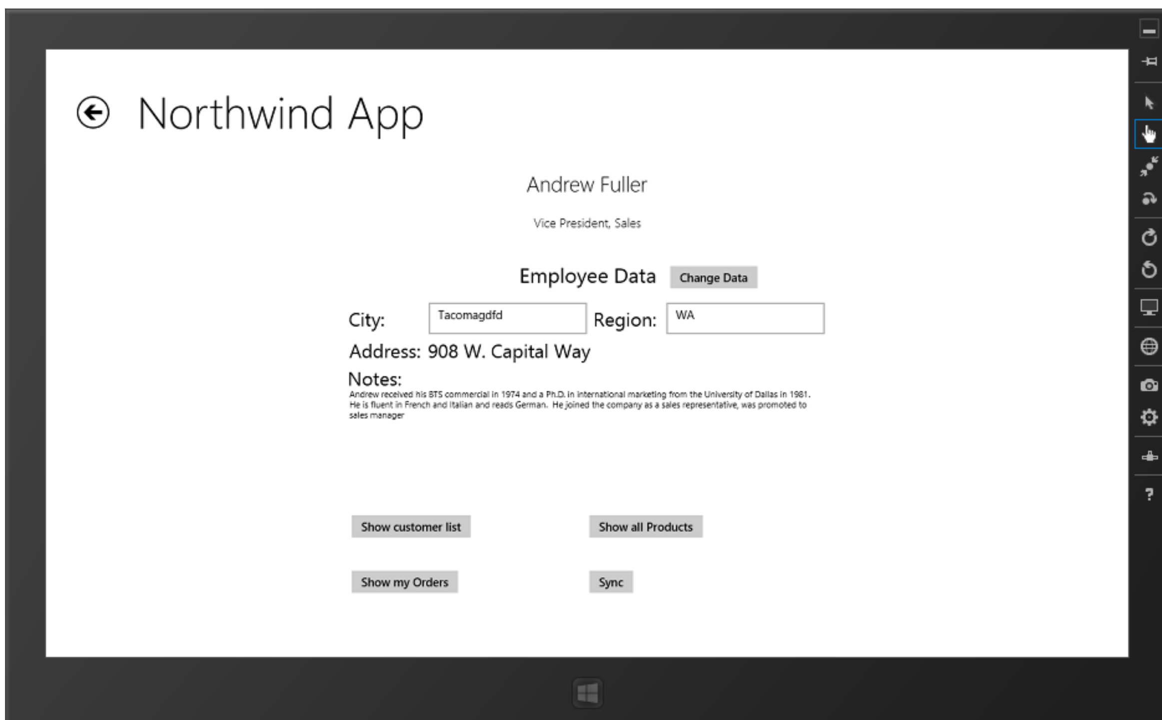


Figure 51: The main view of the sample application on a Windows tablet

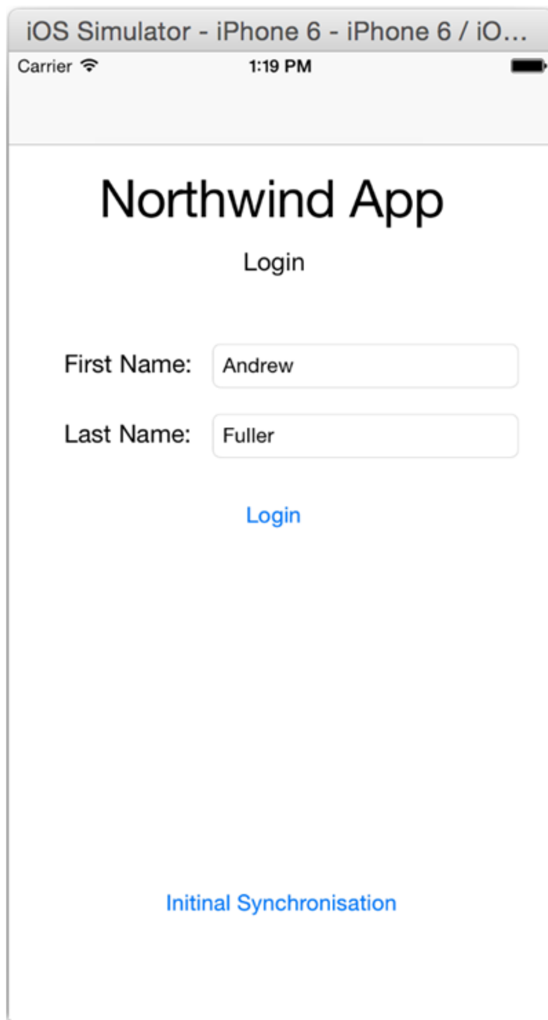


Figure 52: Login page of the iOS sample App

Figure 53: Main Page of the iOS Sample App

By clicking the "Change data" button users can change the city and region of the selected employee, which creates an update event in the database.

By clicking on "sync" on the main page the App establishes a connection to the database server and synchronizes the changes with the central database.

Under the "Show my Orders" menu the Orders of the selected employee is listed.

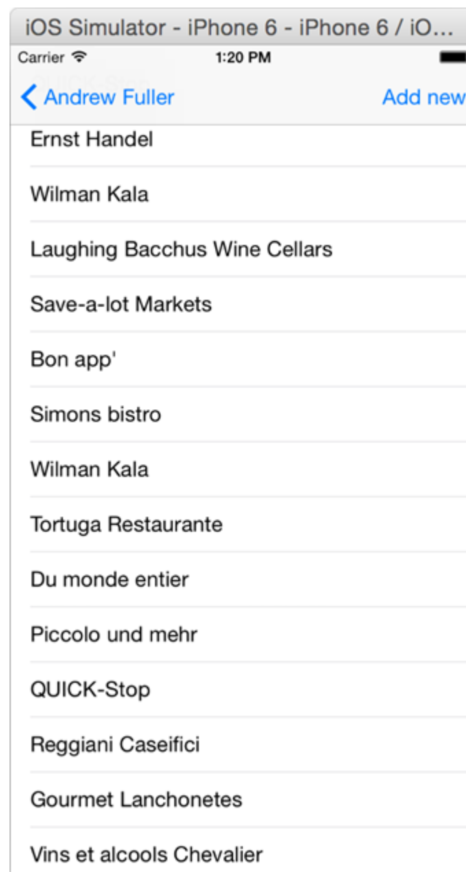


Figure 54: Orders list in the iOS sample App

In this list every row corresponds to an order and the row shows the value of the *ShipName* column of the *Orders* table. By clicking to the "Add new" button users can create a new order. This operation inserts a row into the *Orders* table and another row into the *OrderDetails* table where further details of the created order are stored like the *UnitPrice* and the *Quantity*. The OrderDetails row has a foreign key reference to the *Orders* table.

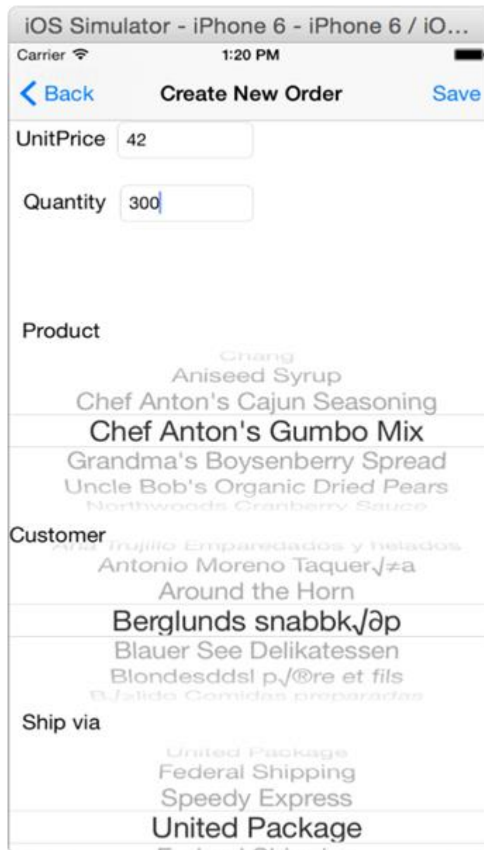


Figure 55: Creating new Order in the sample iOS App

One example for a select operation with MobileSync is when the user clicks the login button. On iOS this method is the following:

```
-(BOOL)shouldPerformSegueWithIdentifier:(NSString *)identifier sender:(id)sender{

    if ([identifier isEqualToString:@"LoginTouchedSegue"]) {
        NSString* firstName = self.FirsNameTextField.text;
        NSString* lastName = self.LastNameTextField.text;

        LocalDbManagerWrapper* localDb = [[LocalDbManagerWrapper
alloc]initWithDbName:@"Northwind"];

        self.employeeTable = [localDb PerformSelect:@"Employees"];

        NSArray* list =  [self.employeeTable GetColumnValues:[NSArray
arrayWithObjects:@"FirstName", @"LastName", nil]];

        for (int i=0; i<list.count; i++) {
            if( [firstName isEqualToString:list[i][0]] && [lastName
            isEqualToString:list[i][1]] ){

                return YES;
            }
        }
        return NO;
    } else {
        return YES;
    }
}
```

First an instance of the *LocalDbManagerWrapper* class is created then a *PerformSelect* massage is sent to the new instance with the "*Employees*" parameter which identifies the name of the table which we want to query. After this in the returned *DataTable* we filter out the unnecessary columns and iterate through the rows and check if there is a person in the row with the same last and first name as the user input.

In the Windows Store App the C# code looks very similar:

```
loginTouched = new RelayCommand(() =>
           {
               var ldbm = new
                MobileSyncWindowsRuntimeComponent.LocalDbManagerWrapper("Northwind");

               ldbm.OpenDb();
               var employees = ldbm.PerformSelect("Employees");

               var namesList = employees.GetColumnValues(new List<string> { "FirstName",
                "LastName" });

               int i = 0;
               foreach (var fullName in namesList)
               {
                   if ((string)fullName[0] == FirstName && (string)fullName[1] ==
                     lastName)
                   {
                       employees.DropAllRowsExcept(i);
                       GlobalDataStore.SelectedUser = employees;
                       navigationService.NavigateTo("MainPage");
                   }
                   i++;
               }
           });
```

First here also a LocalDbManagerWrapper instance is created which opens the Northwind database then it performs a select on the same table. One difference is that here we immediately store the selected employee row in a GlobalDataStore. On IOS this happens in another function.

7. Summary

This section wraps up the work done by this thesis. First it describes the public API of the implemented framework which serves as a documentation for app developers working with MobileSync. The next part of this section goes into the limitations of MobileSync and lists all the possible directions for further development. And at the end a closing section summarizes the findings and consequences.

7.1 The API of MobileSync

This section describes the API of the MobileSync framework. In order to use the implemented solution application developers must include the compiled version or the source code of the MobileSync framework and use the public API in their application to interact with it. This section serves as the documentation of this public interface; it describes the classes and methods which can be used by application developers.

The most important design goal of this public API was simplicity. MobileSync was created to simplify the synchronization related work for application developers and this must be reflected by the API of the framework. This means that everything which is not important for App Developers is hidden. For example change tracking tables are not exposed and the *SyncDetails* table is also completely hidden from developers since these are managed by the system internally and changing them can lead to a state where the framework cannot do its work anymore.
As the requirements in section 1.3 Requirements state the framework does not change the data in the database. It adds additional columns and tables to identify the rows which were changed between two synchronizations, but it never changes ids of tables. The designer of the database has to keep this in mind and design the database in a way that it is able to receive and change data from multiple sources. For example when you chose the id of a table avoid ids where the system uses an integer and the number is always incremented. The problem with this common id type is that when two disconnected client generate a new row into the same table both of them will chose the same id and one of them will never make it to the central database because of id collision. But this collision is not caused by the sync framework itself; it is rather a problem in the database design. So for ids use globally unique identifiers or some other constructs which can deal with these kinds of scenarios.

7.1.1 The iOS API

The hart of the IOS API is the *IosDataTable* class. One instance of this class represents one table in the database. Its first public property is called *Headers* and as the name suggests it represents the header of the database table. This property itself is an array and every item of this array at position i. stores the header details of the column at the position i.
The elements of this array are from type *DataTableHeader* with a *Name* and a *DataType* property.

```
@interface IosDataTable : NSObject
```

```
@property (strong, atomic) NSArray* Headers;
@property (strong, atomic) NSString* Name;
@property (strong, atomic) NSMutableArray* Rows;

-(NSArray*)GetColumnValues:(NSArray*) ColumnNames;

@end

@interface DataTableHeader: NSObject

@property (strong, atomic) NSString* Name;
@property (strong, atomic) NSString* DataType;

@end
```

For Example to get the name of the third column of a table the code looks like this:

```
((DataTableHeader*)myTable.Headers[3]).Name
```

where *myTable* is an instance of the type *IosDataTable*. The *Name* property stores the name of the table. The most important property is called *Rows* which is of type *NSMutableArray*. Every row item stores another array where every item is a field from the database. For example to get the second row from a table you write:

```
myTable.Rows[2]
```

To get the value of the first column (the indexing begins with 0) you write:

```
employees.Rows[2][0];
```

Since the row is stored as an *NSArray* what you get is an *NSObject*, which most of the time you have to cast to the correct type. As it was mentioned before the type information is stored in the header, so to get the name and the type of the column to this field you can use the *Header* property:

```
((DataTableHeader*)myTable.Headers[0]).DataType
```

To manage the local database you can use the *LocalDBManagerWrapper* class

```
@interface LocalDbManagerWrapper : NSObject

-(void)OpenDb;
-(void)CloseDb;
-(void)CreateTable:(IosDataTable*)withTable;
-(void)PerformInsert:(IosDataTable*)onTable andOnRow:(int)rowNumber;
-(IosDataTable*)PerformSelect:(NSString*)onTable;
-(void)PerformDelete:(IosDataTable*)onTable andOnRow:(int)rowNumber;
-(void)PerformUpdate:(IosDataTable*) onTable andOnRow:(int)row;

-(id)initWithDbName:(NSString*) dbName;

-(void)GetAllLocalTableNames;

@end
```

The *CreateTable:(IosDataTable*)withTable* method gets an *IosDataTable* instance and passes it to the C++ layer which stores the table into the local database and creates a change tracking table to the original table (as it was mentioned before this change tracking table is not visible at this layer, it is used only internally by the framework).

Creating a table in the local database works as follows:

```
LocalDbManagerWrapper* dbm = [[LocalDbManagerWrapper alloc]
    initWithDbName:@"TestDbName"];
[dbm OpenDb];
[dbm CreateTable:myTable];
```

The *PerformInsert:(IosDataTable*):onTable andRow(int)rowNumber* inserts a row into the database. The first parameter is the table where the row will be stored and the *rowNumber* is the index of the newly inserted row. This method also takes care of the change tracking as any other method.

As an example here we insert a new row into the *myTable* table:

```
DatabaseRow* newRow = [[DatabaseRow alloc]init];
newRow.columns = [[NSMutableArray alloc] initWithObjects:@"Value1", [NSNumber numberWithInt:2], @"Value3",
    nil];
[myTable.Rows addObject:newRow];
[dbm PerformInsert:myTable andOnRow:myTable.Rows.count-1];
```

The *PerfomSelect:(NSString*) onTable* method returns the whole table with the name *onTable* in an *IosDataTable* instance.

To remove a row from a table there is a *PerformDelete(IosDataTable*)onTable andOnRow:(int)rowNumber* method. This works the same way as the insert method works, the first parameter is the table, the second parameter is the index of the row which will be removed from the table.

Here for example the first row will be removed from the *myTable* table:

```
[dbm PerformDelete:myTable andOnRow:0];
```

In order to update the values in a row first you have to modify the value in the *IosDataTable* and then you can call the *PerformUpdate:(IosDataTable*) onTable andOnRow:(int)row* method, where you put the table itself as the first parameter and the index of the modified row as the second parameter. For example:

```
IosDataTable* selectedTable = [self.localDbManager1 PerformSelect:@"myTable"];
NSString* newValue = @"NewValue";
NSMutableArray* row = selectedTable.Rows[0];
[row replaceObjectAtIndex:1 withObject:newValue];
```

```
[self.localDbManager1 PerformUpdate:selectedTable andOnRow:0];
```

And finally there is one class responsible for the synchronization related tasks and this class is called *SyncManagerWrapper*. This is the only class in the IOS API which triggers network related tasks. This means that the prerequisite for using this class is a stable network connection between the IOS device and the target database.

```
@interface SyncManagerWrapper : NSObject

@property NSString* dbAddress;
@property NSString* remoteDbName;
@property NSString* localDbName;

-(void)StartSync;
-(void)InitDb;

-(id)initWithAddress:(NSString*)address_ andLocalDbName:(NSString*)localDbName_  andRemoteDbName:(NSString*)
remoteDbName_;

@end
```

The designated initializer of the class expects the properties of the local and the remote databases. The first parameter is the address of the remote server as a string (this is typically an IP Address plus a database server name), the second parameter is the name of the local database, which is the name of the database file stored on the IOS device, the last parameter is the name of the remote database on the database server.

For example:

```
SyncManagerWrapper* syncManager = [[SyncManagerWrapper alloc]
initWithAddress:@"192.168.128.194\\SQL2012ENT" andLocalDbName:@"Northwind"
andRemoteDbName:@"NORTHWND"];
```

The rest of the class is very simple: the *StartSync* method triggers the synchronization process between the local and the remote database and the *InitDb* triggers the first initialization. The *InitDb* basically calls immediately into the C++ layer and triggers the *InitDb* method of the *SyncManager* C++ class described in Section 5.3. This means that this method adds the *SyncDetails* table, extends the tables in the remote database with the *RowGUID* column and creates all the local tables and copies the content from the remote database to the IOS device. Both methods generate network traffic, so App developers have to make sure that they are only used when a reliable network connection is established and transferring data is no problem for the users.

Until 2014 summer Objective-C was basically the only language used by App developers working with the IOS platform. On the 2. of June 2014 at the Worldwide Developers Conference (WDC) Apple introduced a new programming language called Swift and from IOS version 8 App developers also can build apps with Swift. (Apple Inc., 2014)

As it is described in 3.4 Calling C++ from Objective-C the interoperability between Objective-C and C++ (and C) is completely supported by the compiler. This is not the case with swift. There is no way to use a C++ class directly in a swift class. Fortunately the interoperability between Objective-C and Swift is completely supported in both directions: Swift classes can use Objective-C classes and Objective-C classes can use Swift classes. In fact all the Objective-C APIs are available from Swift. This means that the Objective-C wrapper introduced in this section can be used in Swift code. The next part of this section focuses on how to call the implemented wrappers from Swift. Since the other direction of the interoperability is not relevant for this topic it is left out from the description.

When a mixed language projected is created in Xcode the IDE automatically adds a Bridging header file to the project which is an .h file. If you want to use an Objective-C class from Swift then its .h file must be included in this bridging header file, otherwise the classes are not visible from the other language. Since the IOS API contains only 3 header files the bridging header will look like this:

```
#import "LocalDbManagerWrapper.h"
#import "IosDataTable.h"
#import "SyncManagerWrapper.h"
```

Of course additionally there can be other header files in the list if the application uses other Objective-C classes in Swift.

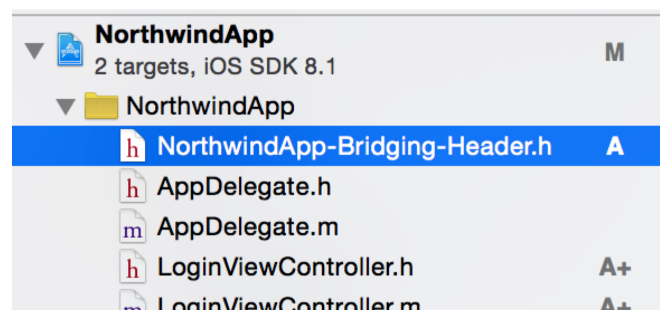


Figure 56: Objective-C - Swift interoperability

When the header files are included into the bridging header the IOS API classes behave like any other classes projected from Objective-C to Swift and the general interoperability rules described in (Apple Inc., 2014) apply to every class.

For the designated initializers the 'initWith' part from the method name is sliced off and every selector becomes a swift function argument. So for example creating a LocalDbMangerWrapper looks like this:

```
let ldbmw = LocalDbManagerWrapper(dbName: “Northwind");
```

The alloc method from Objective-C automatically gets called, so users do not have to take care of allocating the required memory for the object.

The SyncManagerWrapper follows the same principle:

```
let syncManager = SyncManagerWrapper(address: "localhost/SQLServer",      andLocalDbName: "Northwind",
andRemoteDbName: “NORTHWND");
```

When an Objective-C method is converted into Swift the first part of the method name and the rest of the selectors became the parameter list. Every method is called with parentheses. For example selecting a table with the *LocalDbMangerWrapper* instance that we created before works this way:

```
let employeeTable = ldbmw.PerformSelect("Employees");
```

Accessing Objective-C properties works in Swift exactly as it works in Objective-C with the dot notation, so to get the table name you can write:

```
let tableName = employeeTable.Name;
```

The rest of the API in Swift fallows the principles described here.

7.1.2 The Windows API (C#, JavaScript, C++/CX)

The Windows API is very similar to the iOS API. It has the same three main classes with the same methods. One difference is that this API surface is generated by the language projection layer introduced in 3.5 Wrapping C++ code into Windows Runtime Components based on the C++/CX wrapper. In contrast to this for the Objective-C wrapper is everything written manually and there is no code generation involved on the iOS platform.

This means that by writing the C++/CX layer code the framework automatically becomes usable in C#, JavaScript, and in C++/CX (Windows Store Apps can also be written completely in C++/CX).

To reduce redundancy this section shows the API with C# syntax, but it also applies for the 2 other languages. One thing to keep in mind is that the language projection layer also takes the conventions of given the languages into consideration. This means that for example method names begin with lower case letters in JavaScript. But these kinds of things are the only differences; the semantic is the same for every method.

The whole API can be found in the *MobileSyncWindowsRuntimeComponent* namespace.

The LocalDbManagerWrapper class manages the local database and automatically maintains the change tracking tables. The constructor takes a string parameter which is the name of the local database. After an instance is created users can select a table with the *PerformSelect* method which takes the name of the table as a parameter. It also has

methods to insert, delete, and remove rows. For these methods the first parameter is always a *WindowsDataTable* (introduced in this section later) instance which contains the table which will be changed by the method. The second parameter is an integer and it is the zero based index of the modified row.

```
namespace MobileSyncWindowsRuntimeComponent
{
    public sealed class LocalDbManagerWrapper
    {
        public LocalDbManagerWrapper(string dbName);

        public void CloseDb();
        public void CreateTable(WindowsDataTable Table);
        public void OpenDb();
        public void PerformDelete(WindowsDataTable onTable, int rowNumber);
        public void PerformInsert(WindowsDataTable Table, int RowNumber);
        public WindowsDataTable PerformSelect(string TableName);
        public void PerformUpdate(WindowsDataTable Table, int Row);
    }
}
```

The *WindowsDataTable* is the class which represents one table in the database and serves as the first parameter of the delete, insert and update methods in the *LocalDbManagerWrapper* class. The *Headers* property stores the column information for all the columns in the table as a list. It has two properties:

- *Name*: the name of the column
- *DataType*: The datatype of the column

The data itself is stored in the *Rows* property which is a list of lists. Every list item represents a column as a list of objects, so for example to reference the 1. item in the 1. tuple works as follows:

```
myTable.Rows[0][0]
```

The interface of the class is this:

```
namespace MobileSyncWindowsRuntimeComponent
{

    public sealed class WindowsDataTable
    {
        public WindowsDataTable();

        public IList<DataTableHeader> Headers { get; set; }
        public string Name { get; set; }
        public IList<IList<object>> Rows { get; set; }

        public bool DropAllRowsExcept(int rowIndexToKeep);
        public IList<IList<object>> GetColumnValues(IList<string> ColumnNames);
        public object GetValueForRowAndColumnName(int rowNumber, string columnName);
    }
}
```

The *SyncManager* class is responsible for the two synchronization related tasks: 1) The *InitDb* method sets up the local database and if it is necessary it also initializes the remote database. 2) The *Sync* method starts the synchronization between the local and the remote databases. The constructor has 5 parameters and it basically identifies the local and remote databases and it also needs the username and password to the remote database in order to be able to authenticate to the remote server.

```
namespace MobileSyncWindowsRuntimeComponent
{
    public sealed class SyncManagerWrapper
    {
        public SyncManagerWrapper(string localDbName, string remoteDbServerAddress,
       string remoteDbName, string remoteDbUserName, string remoteDbPassword);

        public void InitDb();
        public void Sync();
    }
}
```

As it was mentioned at the beginning of the section the Windows Runtime wrapper is also usable from JavaScript and from C++/CX. On Figure 57: The WinRT Wrapper from JavaScriptFigure 57 a sample is shown where the same Windows Runtime Component is used from JavaScript. As you can see it has the same methods and when these methods are triggered from JavaScript then the Language Projection Layer triggers both in C# and in JavaScript the same C++/CX methods. The only difference here is that the methods names begin with lower case letters. The reason for this is that the convention in JavaScript is that method names are lower case and the Language Projection Layer automatically converts the method names to lowercase names based on this rule.

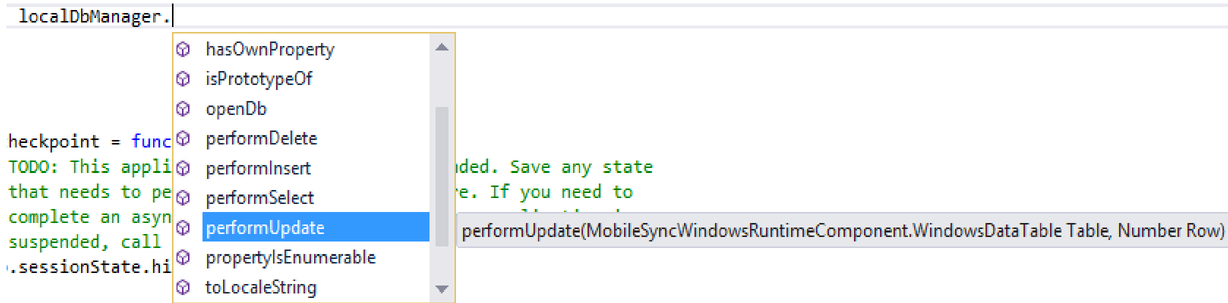


Figure 57: The WinRT Wrapper from JavaScript

7.2 Opportunities for further development

This thesis describes the fundamentals of building a platform independent synchronization framework for mobile devices.

At this point there is an implementation of the proposed solution from Section 4 which is described in this section (Section 5). The framework is able to handle databases with foreign keys, it has a basic conflict resolution functionality built in, and most of the standard database types like the number types, fix and variable length character types, date types are handled by the framework. The current implementation supports only Microsoft SQL Server on the remote side.

1) One direction to further developments would be to implement support for additional remote side databases like Oracle or MySQL. The communication between the framework and the remote database is defined by the IDbConnector interface (Figure 39).

   Support for a new DBMS can be easily added by implementing this interface and by providing it to the *remotedbmanager* instance in the wrapper layers.

2) Another potential feature is the horizontal and vertical filtering of the synchronized database as it is implemented in Microsoft SQL Server (described in 2.2 SQL server data replication).

3) A third area to improve would be the API itself. Currently is uses strings to identify columns which is not ideal since one change in the database can break the application and it is very hard to find the point in the source code which needs to be adapted to the database changes. A solution for this would be to build better ORM capabilities into the framework like it was introduced in 2.4 Related technologies for LINQ in C# and CoreData on the Apple platform.

7.2 Conclusion

This thesis shows how to implement a C++ library for the 3 main mobile platforms which is able to synchronise a SQLite database running inside mobile applications with a classical RDBMS. The reference implementation called MobileSync includes an automatic conflict resolution mechanism based on the "Server wins" approach described in 4.4 The Synchronization process, and it is able to synchronise the local SQLite database to a Microsoft SQL Server.

This reference implementation demonstrates that sharing C++ code between Android, iOS and Windows/Windows Phone is feasible and a very common approach to implement cross platform libraries. Furthermore it also shows that the approach for data synchronisation introduced in section 4. Details of the proposed solution for the synchronization problem can be implemented with the techniques described in section 3.3. Calling C++ from a higher level language.

As an outcome of this thesis we can state that synchronising data between Smartphones/Tablet-PCs and classical DBMSs like Microsoft SQL Server is possible.

The approach with the shared C++ library fullfills all the requirements listed in section 1.3 Requirements:

- It is able to synchronise data between a mobile app and a classical RDBMS (in this case Microsoft SQL Server).
- **Change tracking**: Change tracking is included in the framework and it is based on the row level tracking model of the Microsoft SQL Server data replication feature which was introduced in 2.2 SQL server data replication.
- **Multiplatform**: With the shared C++ library approach and with the wrapping libraries described in section 3.3. Calling C++ from a higher level language MobileSync supports all the major mobile platforms.
- **API fluidity**: Thanks to the approach with the wrapper libaries the framework behaves on every platform as any other framework on that given platform.
- **Automatic conflict resolution** is also included in the framework.
- **Support for SQLite:** Thanks to the native C/C++ API of SQLite the C++ layer can directly communicate with SQLite, so the framework supports SQLite by default.
- **Possible Integratation** with new server side DBMSs is ensured by the *IDBConnector* interface.
- **Simple public API**: The wrapper layers project only the classes and methods to users which are needed to interact with the framework, but nothing more: change tracking and the details of the conflict resolution are completely hidden.
- The framework **does not change the data** stored in the database which it manages.

Another advantage of the solution presented in this thesis is that is has a really small footprint: the synchronization framework is embedded into the client apps running on mobile devices and there is no additional software component involved or necessary to set up the system. This is very unique to this framework: all the solutions presented in section 2. Existing solutions need some additional components. Furthermore the framework is very efficient and maintainable thanks to move semantics presented in 5.2 C++11 move semantic in the DataTable and DataRow classes.

Glossary

C++11:
A C++ standard defined in ISO ISO/IEC 14882:2011. It adds additional capabilities to the previous C++ standards. For example smart pointers, move semantic, range based for loop, lambda expressions. One of the paradigms in the C++ languages started by this standard is in the field of resource management: using C++11 it is possible to avoid row pointers to express ownership in one class.

C++/CX:
A C++ dialect from Microsoft which simplifies the creation of a Windows Runtime Component. Based on some special C++/CX keywords like *ref* the compiler generates COM components which can be used by higher level languages (like JavaScript or C#) in Windows Store and Windows Phone apps.

C#:
An object oriented programming language from Microsoft defined in ECMA-334. The C# compiler compiles the code into a so called Intermediate Language (IL) which runs on the virtual machine called Common Language Runtime (CLR).

CoreData:
An Object-Relational Mapping (ORM) framework from Apple used by application written in Objective-C. This is the main ORM for IOS Apps and it enables App developers to access a SQLite database with strongly typed objects.

CLR:
Common Language Runtime. The execution engine from Microsoft for the platform independent Intermediate Language created by compilers which support the .NET framework like C# and VB.

Dalvik:
A Java Virtual Machine used by the Android Operating System. Normally a JVM runs Java Byte-code. With Dalvik the compilation process of an Android App has an additional step and the Java byte-code is compiled into Dalvik byte code which is the executable for the Dalvik VM.

DBMS (Database Management System):
Software with the purpose to store and retrieve data from/in it. Users and other applications can save and retrieve data from a DBMS. A state of the art DBMS typically supports transactions, it can be accessed by multiple users (multiuser system), it has access control and security features and some kind of backup functionality.

Entity Framework:

An open source Object-Relational Mapping framework from Microsoft for .NET applications.

Entity SQL:

A storage independent query language for the Microsoft Entity Framework. The queries are represented as row strings which are processed by the framework. After the introduction of LINQ to the .NET framework it lost its popularity, since in contrast to LINQ it is not part of the C# and VB languages.

Foreign key:

This refers to a relationship between two tables in a relational database where one row of a table references another row in another table. With a foreign key relationship database designers can force the consistency of a database. For example if there are houses and people stored in a database then a foreign key in the houses table can be defined which points to the people table and defines the pointed person as the owner. In most DBMSs this foreign key relationship is forced, so for example none can remove a person until there is a row in the houses table where this person is the owner of the house.

FreeTDS:

An open source C implementation of the TDS protocol. The code can be downloaded from: http://www.freetds.org/

Java:

An object oriented programming language design by Sun Microsystems. It is used to build enterprise application and it is also the default programming language for Android Apps.

JNI:

Java Native Interface. A part of every major JVM and it defines how Java components can communicate with native components like C functions and C++ classes.

JVM:

Java Virtual Machine: The execution engine for platform independent Java-byte code.

LINQ (Language Integrated Query):

A Domain Specific Languages developed by Microsoft and it enables special syntax to deal with data (like collections, database tables) in C# and Visual Basic as a part of the language.

Microsoft SQL Server:

An RDBMS product from Microsoft.

MobileSync:

The name of the implemented framework in this Master's thesis.

NDK:

Native Development Kit. Built on top of the JNI standard it enables Application developers to

implement some parts of an Android App in native code and the rest of it in Java. NDK enables that Java code can call into the native code.

Objective-C:
An object oriented programming language by apple based on C. It is mainly used to develop IOS and Mac OS applications.

Objective-C++:
With this name we refer to the technology that enables mixing Objective-C code with C++ code. It is by default built into Xcode and you can enable it for a source code file by using the .mm file extension (normal Objective-C code is stored in files with .m extension).

RDBMS:
Relational Database Management System: A DBMS developed by E.F. Codd where the system is based on rows and its relations (also called tables). An RDBMS is for example Microsoft SQL Server or Oracle.

STD (Standard Template Library):
A software library which is defined as a port of the C++ standard. Every major C++ compiler ships with an implementation of the STD. It contains many useful classes like vector, string, and reference counted smart pointers.

SQLite:
An open source embedded DBMS written in C. The project was started by D. Richard Hipp, and today it is the default database management system on Android and IOS.

Swift:
A programming language introduced on the WWDC 2014 by Apple to build IOS and Mac applications. Similar to Objective-C the goal is to make application development easier on Apple platforms. Swift provides some modern features like generic programming, simplification for built in classes like string, optional type names by declaration.

TDS:
Tabular Data Stream. An application layer protocol, originally designed by Sybase. Microsoft also uses it in the SQL Server product to transfer data between the client application and the database server. Its newest version is 7.4 and this version is supported by MS SQL Server 2012 and 2014.

Windows Runtime:
It is an Application Programming Interface written in C++ for the Windows 8 and Windows Phone platforms. Its predecessor was the Win32 API and Microsoft reimplemented some features of the Win32 in Windows Runtime in order to support fast and fluid touch capable applications. By default many of the Windows Runtime APIs are async.

Windows Runtime Component:
A software component which runs inside the Windows Runtime layer and it can be used by other Apps which can be implemented in different high level programming languages like C# or JavaScript

Visual Studio:
An integrated development environment from Microsoft. It has support for multiple programming languages like C#, C++, Visual basic and web development including HTML and JavaScript.

Xcode:
An Integrated Development Environment from Apple. It is mainly used to develop Mac and IOS applications and the current version (6.1) ships with the clang compiler front end.

List of Figures

Figure 1: Classical Smartphone-Database communication 2
Figure 2: MS SQL Data Replication components (MSDN) 7
Figure 3: Selecting Articles in MS SQL Data replication 8
Figure 4: vertical filtering in MS SQL 9
Figure 5: Distribution Database properties in MS SQL 10
Figure 6: Adding Subscription in MS SQL. 11
Figure 7: Communication between the MS SQL Data Replication components. Source: (Sebastian Meine, 2013) 13
Figure 8: Merge replication 15
Figure 9: SQL Server conflict resolver 16
Figure 10: MS SQL synchronization summary 17
Figure 11: MS SQL detailed view of the resolved conflicts 17
Figure 12: MS SQL Tracking level 18
Figure 13: MS SQL subscription type 19
Figure 14: MS SQL Security Properties 20
Figure 15: Oracle GoldenGate components (Gupta, 2013) 22
Figure 16: Oracle setting up logging for GoldenGate 23
Figure 17: The GoldenGate Manager process 24
Figure 18: GoldenGate capture process 25
Figure 19: The CoreData Designer UI in Xcode 33
Figure 20: The proposed architecture 38
Figure 21: SQLite types 40
Figure 22: The windows infrastructure (Microsoft, 2012) 44
Figure 23: Creating a WinRT component in Visual Studio 46
Figure 24: The Android infrastructure (Google inc.) 49
Figure 25: The Android NDK (Gargenta, 2010) 50
Figure 26: Sync initialization step 1: Identify table to initialize 53
Figure 27: Sync initialisation step 2: The RowGUID column is added to the original table and the change tracking table was created, the new SyncDetails table is created as well 54
Figure 28: Content of the SyncDetails table in the initialization step 54
Figure 29: Sync initialization step 3: The server and the client contains the same schema 55
Figure 30: Change tracking for insertion 56
Figure 31: Change tracking after delete operation Case a 57
Figure 32 Change tracking after delete operation Case b 58
Figure 33: Change tracking after delete operation Case c 58
Figure 34 Change tracking after update, Case a 59
Figure 35: Change tracking after update, Case b 60
Figure 36: Change tracking after update, Case c 61
Figure 37: Tbe DatabaseField, Datarow and the DataTable classes 66
Figure 38: The ChangeTrackingRow and ChangeTrackingTable classes 67
Figure 39 The IDbConnector and its implementations 68
Figure 40 LocalDbManager and ChangeTrackingManager 70
Figure 41 The RemoteDbManager class 71
Figure 42 The SyncManager class 73
Figure 43 The UML Diagram of the MobileSync C++ layer 75

Figure 44: C++ container 76
Figure 45 Copy constructor in collections 77
Figure 46 Returning a container without move semantics in C++.. 78
Figure 47: C++ move semantics step one: doe not destroy references from the temporary object 79
Figure 48: C++ move semantics step two: steal the pointers from the temporary container... 79
Figure 49: The Northwind Database (Microsoft).. 87
Figure 50:The login page of the sample application on a Windows Tablet.. 88
Figure 51: The main view of the sample application on a Windows tablet 88
Figure 52: Login page of the iOS sample App.. 89
Figure 53: Main Page of the iOS Sample App.. 90
Figure 54: Orders list in the iOS sample App 91
Figure 55: Creating new Order in the sample iOS App 92
Figure 56: Objective-C - Swift interoperability 98
Figure 57: The WinRT Wrapper from JavaScript.. 101

References

Alex Allain, A. T. (2014, September 7-12). Practical Cross-Platform Mobile C++ Development at Dropbox. Bellevue, WA, United States of America: CppCon 2014.

Allen, G. (2010). *The Definitive Guide to SQLite.* New York: Apress.

Apple inc. (n.d.). *Apple Developer*. Retrieved 2014, from https://developer.apple.com/

Apple inc. (n.d.). *CoreData*. Retrieved 12 20, 2014, from https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html

Apple inc. (n.d.). *LocalizedStandardCompare*. Retrieved 12 20, 2014, from https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/index.html#//apple_ref/occ/instm/NSString/localizedStandardCompare:

Apple Inc. (2014). *The Swift Programming Language.* Cupertino.

Apple Inc. (2014). *Using Swift with Cocoa and Obejtive-C.* Cupertino.

Christian Nagel, J. G. (2014). Professional C# 5.0 and .Net 4.5.1. Indianapolis: Wrox.

Gargenta, M. (2010, 02 23). Learn about Android Internals and NDK .

Gartner. (2014). *Gartner Says Sales of Smartphones Grew 20 Percent in Third Quarter of 2014* . Egham, UK: Gartner.

Gartner. (2014). *Gartner Says Worldwide Tablet Sales Grew 68 Percent in 2013, With Android Capturing 62 Percent of the Market.* Egham, UK: Gartner.

Google inc. (n.d.). *Android Developers*. Retrieved 2014, from http://developer.android.com/index.html

Gupta, A. (2013). *Oracle GoldenGate 11g Complete Cookbook.* Packt Publishing.

Hegarty, P. (2013-14 Fall Semester, Lecture 13). CS 193P iPhone Application Development. Stanford : Stanford University.

Microsoft. (n.d.). *Northwind*. Retrieved 2015, from Northwind: https://northwinddatabase.codeplex.com

Microsoft. (2012, 2 12). Windows 8 and the future of XAML: The Windows Runtime (WinRT).

MSDN. (n.d.). *MSDN*. Retrieved 11 17, 2014, from http://technet.microsoft.com/en-us/library/ms165654%28v=sql.90%29.aspx

MSDN. (n.d.). *Windows Runtime C++ Template Library (WRL)*. Retrieved 01 2015, from Windows Runtime C++ Template Library (WRL): http://msdn.microsoft.com/en-us/library/windows/apps/hh438466(v=vs.110).aspx

Oracle FAQs. (n.d.). *Oracle FAQs*. Retrieved 1 3, 2015, from http://www.orafaq.com/wiki/Oracle_Streams

Oracle Streams. (n.d.). *Overview of Oracle Streams Replication*. Retrieved 01 02, 2015, from http://docs.oracle.com/cd/B28359_01/server.111/b28322/gen_rep.htm#i1007907

Sebastian Meine, P. (2013). *Fundamentals of SQL Server 2012 Replication.* Simple Talk Publishing.

sqlite.org. (n.d.). *Datatypes In SQLite Version 3*. Retrieved 2014, from https://www.sqlite.org/datatype3.html#affinity

sqlite.org. (n.d.). *sqlite.org*. Retrieved 2014, from http://sqlite.org/

Tatjana Stankovic, S. P. (2010). Platform Independent Database Replication Solution Applied to Medical Information System. *14th East-European Conference on Advances in Databases and Information Systems* . NoviSad: Springer.

Thomas Becker, P. (2013). *C++ Rvalue References Explained* .

Tony Antoun, I. Z. (2014, September 7-12). How Microsoft Uses C++ to Deliver Office (and More) Across iOS, Android, Windows, and Mac, Part I, Part II . Bellevue, WA, United States of America: CppCon 2014.

Biography

Personal Details

Name:	Gergely Kalapos
Date of Birth, Place:	*****************
Nationality:	Hungarian
Place of residence:	****************
Phone:	****************
E-Mail:	****************

Education

Johannes Kepler University (Linz, Austria)
M.Sc. (Dipl.-Ing.), Software Engineering Since 2013

Pazmany Peter Catholic University, Faculty of Information Technology (Budapest, Hungary)
B.Sc., Computer Engineering - major in Software Engineering (2013)
Thesis: http://users.itk.ppke.hu/~kalge/.thesis/Szakdolgozat_Kalapos_Gergely_0412.pdf

Languages

German	State Accredited Language Exam (B2 level)
English	State Accredited Language Exam (B2 level)
Hungarian	Native

Employment History

***** 2014 - Current: ************* ******, Austria
Software Engineer

**** 2012 - ***** 2014 *********, Austria
C#/.NET Software Developer

**** 2008 - ****** 2009: IT-Services Hungary, Budapest
German speaking service desk agent

Summer 2005, 2006: Siemens PSE Kft., Summer 2007: Siemens Zrt. Budapest
Intern

Certifications

MCTS: Web Applications Development with Microsoft .NET Framework
Microsoft transcript:

https://mcp.microsoft.com/Anonymous//Transcript/Validate
Transcript ID: ******* Access Code: ********

Skills

C#:	I started C# in 2009. I completed several university projects in C# and since 2012 I write production code in C#.
Java:	I started Java in 2011 spring semester, and I developed a basic FTP server as a final project for a Java course. Since then I am a seasonal Java developer, I use the language mostly on university and private projects.
C++:	I coded in C++ first at High School. As a part of an advanced C++ course I developed in a team a 2D network game with the ClanLib SDK.
Other languages:	Objective-C, JavaScript.
Database skills:	I plan and implement databases for my applications and I am able to write Triggers and Stored procedures and query databases with SQL. I have experience with Oracle and Microsoft SQL server.
Design Patterns:	GoF patterns, MVC, MVVM
IDEs:	Visual Studio 2010, 2012, 2013, SQL Management Studio, Eclipse, Xcode
Additional:	SVN, Team Foundation Server, Unit testing

Hobbies

- Running. (Half Marathon time: 1:38:00 Verona, Italy, 2014)
- Stock Exchange. With my friends we won Morgan Stanley's stock exchange simulation game, called "Portfolio in Peril" in 2009.