Efficient Data Synchronization in Mobile Applications

by

Sebastián Salazar Tapia

A thesis submitted in partial fulfilment for the degree of

MASTER OF SCIENCE

in

Software Engineering

Supervised by:

Dr. Dimitris Kolovos

UNIVERSITY OF YORK

2014

Number of words = 37,741 as counted by the MS Word word count command. This includes all the body of the report and but excludes appendixes.
ABSTRACT

Mobile devices are becoming the main mean of Internet accessing. A great number of these applications allow users to capture, store and consume rich content. A wide variety of mobile applications are data-driven in nature, and users expect that their applications and data will be available anytime and anywhere, regardless of the availability and quality of network connections. In order to keep application responsiveness, local storage is used to capture data while the application is offline. Then, when connectivity is available, data can be synchronized to central repositories in order to backup this data or share it with other users.

Nevertheless, the implementation of synchronization functionalities requires the definition of complex architectures and algorithms that are difficult to implement, dramatically extending the effort and time needed to produce even simple data centric applications. Usually, mobile application developers have to deal with these difficulties in each project even though they have similar requirements.

This project presents a tool designed to solve these problems. SyncAndroid Systems offer an alternative to rapidly develop data centric mobile applications with synchronization capabilities through the use of an API which abstracts most of the challenges that this kind of projects must deal with.
ACKNOWLEDGEMENT

I would like to thank my supervisor Dimitris Kolovos who provided me with invaluable help and guidance through the project.

Finally, I would like to express my gratitude to my parents because all my achievements are for them and thanks to them. [Gracias a mis padres, porque todos mis logros los he alcanzado gracias a ellos.]

STATEMENT OF ETHICS

The project has been carried out in an ethical manner, with all respective research referenced and all assistance acknowledged. No significant ethical implications of the project outcomes could be identified.
# TABLE OF CONTENTS

ABSTRACT ............................................................................................................................ I
ACKNOWLEDGEMENT .......................................................................................................... II
STATEMENT OF ETHICS ....................................................................................................... II
TABLE OF CONTENTS .......................................................................................................... III
LIST OF TABLES .................................................................................................................... V
LIST OF FIGURES ................................................................................................................. V
LIST OF LISTINGS ............................................................................................................... VI

CHAPTER 1  INTRODUCTION ............................................................................................. 1
  1.1. BACKGROUND ............................................................................................................. 1
  1.2. PROJECT GOAL .......................................................................................................... 2
  1.3. REPORT STRUCTURE ............................................................................................... 2

CHAPTER 2  LITERATURE REVIEW .................................................................................... 3
  2.1. PERSISTENT DATA STORAGE OPTIONS IN ANDROID ........................................... 3
      2.1.1. Shared Preferences ............................................................................................. 3
      2.1.2. Internal Storage ................................................................................................ 4
      2.1.3. External Storage ............................................................................................... 4
      2.1.4. SQLite Databases ............................................................................................. 4
      2.1.5. Network connection ....................................................................................... 5
      2.1.6. Conclusions ..................................................................................................... 5
  2.2. DATA SYNCHRONIZATION .................................................................................... 5
      2.2.1. Layers of synchronization ............................................................................... 6
      2.2.2. Synchronization Strategies ............................................................................. 6
      2.2.3. Synchronization protocols .............................................................................. 7
      2.2.4. Conclusion ...................................................................................................... 9
  2.3. SQL DATABASES AND NOSQL DATABASES ....................................................... 10
      2.3.1. Relational Databases and SQL ....................................................................... 10
      2.3.2. NoSQL Databases ........................................................................................... 12
      2.3.3. Couchbase and Couchlite ............................................................................. 18
  2.4. MODEL DRIVEN SOFTWARE DEVELOPMENT (MDS) ....................................... 22
      2.4.1. The Model Driven Architecture (MDA) ......................................................... 22
      2.4.2. Conclusions ................................................................................................... 23
  2.5. DOMAIN SPECIFIC LANGUAGES ........................................................................ 23
      2.5.1. Conclusions .................................................................................................... 24
  2.6. INTEGRATED DEVELOPMENT ENVIRONMENTS (WTF, ADT AND EPSILON) .... 25
      2.6.1. Conclusions ................................................................................................... 25
  2.7. SUMMARY ............................................................................................................... 26

CHAPTER 3  METHODOLOGY .............................................................................................. 27
  3.1. PLAN-DRIVEN METHODOLOGIES AND AGILE METHODOLOGIES COMPARISON ........................................... 27
  3.2. PLAN-DRIVEN METHODOLOGY SELECTION ................................................... 28
  3.3. SUMMARY ............................................................................................................... 29

CHAPTER 4  REQUIREMENTS ............................................................................................... 30
  4.1. REQUIREMENTS ELICITATION ............................................................................. 30
      4.1.1. Statement of Need (SON) ............................................................................... 30
      4.1.2. Stakeholder identification ............................................................................... 30
      4.1.3. Stakeholder requirements ............................................................................... 31
      4.1.4. Conclusion ...................................................................................................... 31
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.4.</td>
<td>System Requirements</td>
<td>32</td>
</tr>
<tr>
<td>4.1.5.</td>
<td>Requirements Traceability Matrix</td>
<td>35</td>
</tr>
<tr>
<td>4.2.</td>
<td>REQUIREMENT ANALYSIS</td>
<td>35</td>
</tr>
<tr>
<td>4.2.1.</td>
<td>Use case model</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2.</td>
<td>Use Case Description</td>
<td>37</td>
</tr>
<tr>
<td>4.3.</td>
<td>SUMMARY</td>
<td>37</td>
</tr>
<tr>
<td>5.</td>
<td>DESIGN</td>
<td>38</td>
</tr>
<tr>
<td>5.1.</td>
<td>DESIGN PROCESS</td>
<td>38</td>
</tr>
<tr>
<td>5.2.</td>
<td>ARCHITECTURE</td>
<td>39</td>
</tr>
<tr>
<td>5.2.1.</td>
<td>Quality Attributes and Tactics</td>
<td>39</td>
</tr>
<tr>
<td>5.2.2.</td>
<td>High Level Architecture</td>
<td>41</td>
</tr>
<tr>
<td>5.2.3.</td>
<td>Component Design</td>
<td>43</td>
</tr>
<tr>
<td>5.3.</td>
<td>SUMMARY</td>
<td>56</td>
</tr>
<tr>
<td>6.</td>
<td>IMPLEMENTATION</td>
<td>57</td>
</tr>
<tr>
<td>6.1.</td>
<td>PREPARATION</td>
<td>57</td>
</tr>
<tr>
<td>6.1.1.</td>
<td>IDE Preparation</td>
<td>57</td>
</tr>
<tr>
<td>6.1.2.</td>
<td>Couchbase Lite Tools</td>
<td>57</td>
</tr>
<tr>
<td>6.2.</td>
<td>COMPONENTS IMPLEMENTATION</td>
<td>58</td>
</tr>
<tr>
<td>6.2.1.</td>
<td>API Generator implementation</td>
<td>58</td>
</tr>
<tr>
<td>6.2.2.</td>
<td>SyncAPI Component</td>
<td>63</td>
</tr>
<tr>
<td>6.2.3.</td>
<td>Central Repository</td>
<td>67</td>
</tr>
<tr>
<td>6.3.</td>
<td>SUMMARY</td>
<td>70</td>
</tr>
<tr>
<td>7.</td>
<td>EVALUATION</td>
<td>71</td>
</tr>
<tr>
<td>7.1.</td>
<td>TESTING</td>
<td>71</td>
</tr>
<tr>
<td>7.1.1.</td>
<td>Testing Approach</td>
<td>71</td>
</tr>
<tr>
<td>7.1.2.</td>
<td>Test Case specification and Logs</td>
<td>71</td>
</tr>
<tr>
<td>7.1.3.</td>
<td>Test Summary</td>
<td>76</td>
</tr>
<tr>
<td>7.2.</td>
<td>CASE STUDY</td>
<td>77</td>
</tr>
<tr>
<td>7.2.1.</td>
<td>Evaluation Criteria</td>
<td>77</td>
</tr>
<tr>
<td>7.2.2.</td>
<td>Demonstrator Application</td>
<td>77</td>
</tr>
<tr>
<td>7.2.3.</td>
<td>Usability</td>
<td>78</td>
</tr>
<tr>
<td>7.2.4.</td>
<td>Performance</td>
<td>81</td>
</tr>
<tr>
<td>7.2.5.</td>
<td>Case Study Summary</td>
<td>82</td>
</tr>
<tr>
<td>7.3.</td>
<td>SUMMARY</td>
<td>83</td>
</tr>
<tr>
<td>8.</td>
<td>CONCLUSION</td>
<td>84</td>
</tr>
<tr>
<td>8.1.</td>
<td>PROJECT REVIEW</td>
<td>84</td>
</tr>
<tr>
<td>8.2.</td>
<td>FUTURE WORK</td>
<td>85</td>
</tr>
<tr>
<td>8.2.1.</td>
<td>Queries</td>
<td>85</td>
</tr>
<tr>
<td>8.2.2.</td>
<td>Security</td>
<td>85</td>
</tr>
<tr>
<td>8.2.3.</td>
<td>Support for Custom Indirect Authentication</td>
<td>85</td>
</tr>
<tr>
<td>8.2.4.</td>
<td>Auto-fixing in Model Validation</td>
<td>85</td>
</tr>
<tr>
<td>8.3.</td>
<td>PERSONAL GROWTH</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>APPENDIX</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>TABLE OF APPENDICES</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>APPENDIX A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>APPENDIX B</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>APPENDIX C</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>APPENDIX D</td>
<td>J</td>
</tr>
</tbody>
</table>
Figure 6-7. Spring MVC Restful Service Architecture Diagram.[130], [131]................................. 67
Figure 6-8. SyncGateway Start Command............................................................................. 69
Figure 6-9. Couchbase Server Console .............................................................................. 69
Figure 7-1. ToDo Lite Design............................................................................................... 78
Figure 7-2. Performance Tests............................................................................................. 82
Figure 7-3. Performance Tests............................................................................................. 82

LIST OF LISTINGS

| Listing 5-1. DSL ToDo Example .......................................................... | 44 |
| Listing 6-1. New SyncAndroid Wizard Configuration ................... | 59 |
| Listing 6-2. ToDo Model Example .................................................. | 60 |
| Listing 6-3. EVL Constraint example. ........................................... | 61 |
| Listing 6-4. Coordination Rules of EGL Templates ....................... | 62 |
| Listing 6-5. Generation Template for Setter Method of References | 63 |
| Listing 6-6. DataAPI Use................................................................. | 64 |
| Listing 6-7. User Manager API Usage Example ............................. | 65 |
| Listing 6-8. Use of Activities Provided by SyncADocLib Library ...... | 66 |
| Listing 6-9. Use of Synchronization Manager API ......................... | 66 |
| Listing 6-10. Web Service Handler Definition ............................... | 68 |
| Listing 6-11. Web Service Exception ............................................ | 68 |
| Listing 6-12. userExists Method ................................................... | 69 |
| Listing 6-13. Application Server Configuration ............................. | 69 |
| Listing 6-14. Android Application Configuration ........................... | 70 |
| Listing 7-1. Hand Coded Task Creation ......................................... | 80 |
| Listing 7-2. Task Creation Using Data API .................................... | 80 |
Mobile devices are becoming the main mean of Internet accessing. A recent report By Cisco [1] informs that from 2012 to 2013 the number of mobile devices increased over half a billion and it is expected that the number of mobile devices will reach the huge number of 8.2 billion by 2018 worldwide. The usage of these mobile devices is driven mainly through applications which are made available to users through marketplaces. Google market offers more than 1’300.000 applications and only in July of 2014 almost 35.000 new applications were added [2]. A great number of these applications allow users to capture, store and consume rich multimedia content. The amount of data that these applications are creating has exploded and the ability to make this data available to users any time and everywhere has become essential to the value of mobile applications.

1.1. Background

Mobile applications manage a wide variety of data which have evolved from just text and numbers-bases data to rich multimedia content. A wide variety of mobile applications are data-driven in nature, and users expect that their applications and data will be available anytime and anywhere, regardless of the availability and quality of network connections. Mobile applications like social media platforms, productivity and collaboration applications can become unresponsive whether it depends on network connectivity to operate because even using the best tools and technology, problems like intermittent connectivity and high latency of network access cannot be avoided. This makes essential for mobile applications to offer offline support.

In order to keep applications responsiveness regardless of network connectivity, local storage is used to capture data while the application is offline. This allows user to use their application accessing, creating and modifying their data even when the network connectivity is not available or its latency is high. Then, when connectivity is available, data can be synchronized pushing new data, data which have changed and pulling data created in other devices.

Nevertheless, the implementation of synchronization functionalities requires the definition of complex architectures and algorithms that are difficult to implement extending dramatically the effort and time needed to produce even simple data centric applications. Usually, mobile application developers have to deal with these difficulties in each project even though they have similar requirements: manage data exchange, deal with network failures, propagate changes and detect and resolve conflict. This is why Agrawal et al. [3] conclude that “Expecting every developer to be an expert at building infrastructure for data syncing is nor ideal. Mobile developers should be able to focus on implementing the core of apps”.

This project presents a tool designed to solve these problems. SyncAndroid Systems presents an alternative to rapidly develop data centric mobile applications with synchronization capabilities through the use of an API which abstracts most of the challenges that this kind of projects must deal with.
1.2. Project Goal

The aim of this project is to implement a model-driven solution for generating efficient and correct-by-construction data-synchronisation layers for Android applications. In particular, the objectives of the project are to:

- Design a Domain Specific Language that enables engineers to model the data managed by a mobile application;
- Design and implement a model-to-text transformation that generates a server-side API through which client apps can interact with the server-side data storage;
- Design and implement a model-to-text transformation that generates a client-side API through which Android applications can manage their local data;
- Design and implement a layer through which the local data can be efficiently synchronised with remote data by orchestrating the server-side and client-side APIs.

The results of the project will be evaluated through a demonstrator application.

1.3. Report Structure

The report summarises the whole development process of this project. Chapter 2 reviews the literature in areas related to this project emphasising on in topics like data synchronization and NoSQL databases which were decisive for the final product of the project. Chapter 3 provides a discussion about the development methodologies considered for this project and argues the motivation for selecting the Rapid Application Methodology. Chapter 4 summarizes the requirement elicitation and analysis process, providing a list of functional and non-functional requirements followed by the proposed system requirements. Chapter 5 argues the design process adopted for this project and then presents the design of the system. Chapter 6 summarizes the implementation process and provides a brief review of how the product can be used. Chapter 7 goes through the testing process and evaluates the final product through the implementation of a demonstrator application. Finally, 0 summarizes the project and identifies possible areas for further work.
Mobile computing gives users the chance to access data that is stored in a stationary database or some other data repository at any time and any place. This is implemented through the use of modern communication links like wireless networks, cellular networks, Bluetooth connections and others, which allow users to capture, store, share and consume multiple kinds of rich content. Common examples of mobile devices are laptops, but also smaller devices like PDAs or smart cell phones. Mobile computing is regarded by many researchers as the worst case of distributed computing [4]–[6], since it is constrained by many factors that have to be faced and overcome in order to provide reliable and trustworthy data-centred applications to end-users [6]. The main constraints that have to be taken into account in the context of this project are:

- Resource-limitations of mobile devices compared to static computers
- Mobile connectivity is highly variable in performance and reliability. These factors can trigger/cause disconnected and failed operations.

Users of mobile devices, also known as nomadic users [6], require local copies of their data, since they are often disconnected from their home or work network, making support to offline operation important. But local data store is not enough, applications should provide the facility to synchronize or replicate its data as soon as they are back online. This gives user the capability to back up their data in safer places than a mobile device and the capability to manipulate their data from multiple devices.

Implementing applications prepared for offline support is not easy. It involves complex synchronization models and algorithms that should be designed to operate in an unstable network environment with variable performance. This presents challenges that are difficult to face.

This chapter analyses the existing literature in the areas of data synchronization and how it can be applied in the context of Android mobile applications. The first section briefly reviews the data storage options that the Android operative system provides to developers. It is followed by an analysis of data synchronization solutions that have been applied in different mobile applications. As data synchronization mechanisms are also complex to implement for relational databases, the next addressed topic is NoSQL databases and how they can be used in the context of this project. Then, a review of Model Driven Software Development is presented, as well as how it can be implemented with Model Driven Architecture. Additionally, the Domain Specific Language concept is reviewed to analyse the benefits that this can provide to the project. Finally, the rationale used to choose modelling tools and the integrated developing environment is analysed.

### 2.1. Persistent Data Storage Options in Android

Android provides different options for data storage which are supported by the operative system. The choice among these options depends on the particular needs of the application to be developed. One of the decisive factors here is the need for sharing data and the size of the data to be managed [7]. The storage options natively supported by android are presented in this section.

#### 2.1.1. Shared Preferences

Shared preferences are a dictionary of key value pairs of primitive data that can be saved and retrieved using the SharedPreferences class. The data stored using this method will persist across user sessions, making this method ideal for storing small amounts of data like application settings and preferences.
Shared preferences provide support to store primitive data types only. If complex data, like objects is stored, those objects should be serialized. Nevertheless, this is not the recommended use case for shared preferences [7]. Changes in data are made as an atomic batch, meaning they happen as a single operation when changes are committed. This allows to maintain the integrity of the data and roll back changes if the operation fails.

As its name suggests, this storage method is best suited for storing small and simple data. Trying to use this method for storing sophisticated data structures can be overly difficult and will be limited by its obvious constraints.

2.1.2. Internal Storage

For internal file storage, Android designates a private directory to every application where it can store files. This directory is outside of the application bundle and cannot be accessed by other applications or by the user.

The internal storage is a standard file system. Therefore, the developer is responsible for serializing and deserializing the data within those files. It is also important to notice that these files are linked to the lifetime of the applications [7]. Consequently, they are deleted when the application is uninstalled from the device, but they are maintained between updates.

2.1.3. External Storage

As internal storage, this is a standard file system, but the files in these directories can be manipulated by other applications or directly by the user. External storage can be located in external SD cards or in a separate partition on internal storage of the device.

There are two main reasons to use external storage over internal storage. First, if the data is to be public or shared with other applications, external storage should be used. The second reason is space requirements. Usually, external storage is times bigger than internal storage [7].

These files can also be mounted by other devices such as computers when the Android device is connected to them. While the external storage is mounted in other devices or when the media is removed (like SD cards), it can become unavailable.

2.1.4. SQLite Databases

Android provides full support for SQLite databases without imposing any limitations beyond the standard SQLite concepts [7]. SQLite is a C based relational database whose main design goal is simplicity. It is simple to administer, operate, embed in a larger program and to maintain and customize [8].

These characteristics make SQLite widely used in embedded systems. Also, as SQLite is serverless, it does not run its own process. It rather runs in the same process as the application that consumes it and is stored in a single file in the device making it easy to be consumed by Android applications.

SQLite implements most of the SQL standards, making it easy to be adopted by developers who are familiar with other high powered databases. In order to achieve simplicity, SQLite sacrifices other characteristics, such as high concurrency, fine-grained access control, a rich set of built-in functions, stored procedures, esoteric SQL language features, XML and/or Java extensions, tera- or peta-byte scalability, and others [8].

This storage option is the appropriate one when a complex data structure is to be persisted. Nevertheless, the relational model used in SQLite also raises challenges for data synchronization and replications. These challenges will be discussed in sections 2.2 and 2.3.1.
2.1.5. Network connection

The last storage method mentioned by the official Android developer’s documentation is network connection. An Android device using a network connection can connect to different services to exchange and persist data, giving developers almost unlimited options to store data in different formats and locations. However, its main limitation is its dependency on connection availability.

Mednieks claims that “network connection is not really a data storage technique” [9, p. 125] but it can be a way of persisting data outside the bundle of the device. An Android device using a network connection can connect directly to online databases, web services or other devices to interchange and persist data in virtually any kind of formats.

As mentioned before, the main limitation of this method depends on the network connection availability. Improvement in network communication in mobile environments provides developers the opportunity to support collaborative work [4] but this scenario also leads developers to face new challenges when communicating with remote services. Most of the task that can be easily solved in a local environment can lead to different difficulties in a network environment. This is especially evident in a mobile environment. The main causes of these pitfalls are that the network connection is not always present, its availability and communication capability are usually unpredictable [10] and mobile network is loosely coupled. These particularities make the use of this method unsuitable if it is not combined with one or more of the methods mentioned before in order to keep data availability even when the network connection is lost.

2.1.6. Conclusions

Most Android applications need to save data and some of them must manage large amounts of information in files and databases. To fulfil these requirements, Android natively supports the options discussed above which provide Android developers several options to save persistent application data. The choice between these options depends on the specific needs of the application under development, such as how much space the data requires and whether it should be private to the application and in some cases developers can combine these options. Even through these storage options are widely supported by Android, none of them provides synchronization facilities. Therefore the next topic to be analysed is how the data contained in Android applications can be replicated to other devices and a central repository.

2.2. Data Synchronization

Nowadays, when more than half of the global population owns a mobile phone [11] but network services are not always available, often mobile applications need to keep a physical copy of the data the device. Cellular connection services can be sporadic inside buildings and vehicles, and Wi-Fi connections are not available in all locations. Additionally, wireless access can be blocked and downtime can be encountered [12]. These issues lead to the necessity of maintaining a local repository of data on the device, together with the capability of synchronizing this data when network services become available.

To keep multiple copies of data in different databases, two different concepts can be applied: database replication and database synchronization. Database synchronization and replication concepts are closely related and sometimes these terms are used interchangeably [13] but they target two different problems.

Replication is commonly used to keep identical copies of the complete data set on two or more instances. This solution is used to provide high availability and performance. In this scheme, databases instances can work independently and the relationships among them are equal and symmetric. Typically, the number of DB instances is small.

In contrast, in a synchronization context, a central database assumes the role of the server while two or more databases residing on different devices assume the role of clients composing a client-server system. The central database contains copies of the data contained in all the devices, while each local database in
the devices only keeps private data and some shared data. Clearly, synchronization is the concept to be applied in this project because the server side of our solution should serve several devices or clients while each device should maintain a small subset of data owned by the user of each device.

In more detail, synchronization is the “process of establishing consistency among data from a source to a target data storage and vice versa and the continuous harmonization of the data over time” [14]. Synchronization can take place between two or more systems where there is always one active system that controls the synchronization process. A system can explicitly start the synchronization process, which is called push synchronization or pull synchronization where a system is being asked to engage in a synchronization process [15].

In this section some fundamental concepts of synchronization will be discussed. Then the main synchronization strategies will be presented and how these strategies are implemented using synchronization protocols in solutions provided by different authors.

2.2.1. Layers of synchronization
Stage [15] defines two layers of synchronization. The first layer concerns the synchronization of files among file systems. A second layer is defined for the synchronization of data items which are stored in databases.

Android provides different options for data storage which are within the bundle of the first layer. Internal and external data storage options are based in the file system of the operative system [7]. Usually files that do not compromise data structure within themselves are stored using these options. Version control is usually enough to synchronize this kind of data items.

On the other hand, the second layer concerns synchronization within databases that is better known as replication. Android provides the option of using a relational database to persist structured data.

As mobile application also manages rich multimedia data composed by both, structure data and files, and ideal solution should provide the capability of synchronization for both layers.

2.2.2. Synchronization Strategies
The fundamental issue in data synchronization is how to replicate updates to multiple copies of the same data item without causing version and consistency errors. To solve this issue different strategies can be applied. These strategies are usually divided into two wide categories: optimistic and pessimistic [18].

Pessimistic synchronization avoids update conflicts by restricting updates to a single replica based on the presumption that update conflicts are probable. This makes this approach suitable for local area networks in which latencies and availability problems are not frequent. However, it is unsuitable for mobile environments where communication availability between nodes cannot be guaranteed. An example of this kind of strategy is the synchronous replication model.

In contrast, optimistic synchronization strategies allows propagation of data changes concurrently in several nodes assuming that update conflicts are not frequent. In these cases conflicts should be detected and resolved after they occurred. This approach provides higher availability, while it also can lead to data conflicts. The use of this approach in mobile environments has been studied in several research efforts [14], [15], [19]–[21]. Examples of this category are asynchronous, snapshot and push and pull strategies.

A. Synchronous Data Synchronization
Under synchronous data synchronization models, updates are applied to all the databases of the system as part of one atomic transaction. The main advantage of this model is that it prevents stale data and guarantees structural consistency. Consistency is maintained because, if a transaction tries to write to all

---

1 A data item is an atomic state of a particular object at a certain point in time.[16], [17]
databases of the system and one is not available, the transaction will be rejected [22]. This has a cost, this model does not provide availability advantages and network unavailability prevents transaction execution. Therefore, this is model unsuitable for mobile environments where network availability and performance cannot be guaranteed [18].

B. Asynchronous Data Synchronization

For mobile environments, asynchronous synchronization is commonly a better option. Under asynchronous replication, update operations are done at one database at a time, and after it is performed, the transaction is propagated to other nodes [22]. The main advantage over the synchronous model is that replication can be postponed if network connection is not available. Therefore, this model demands less networking and hardware resources than synchronous replication. However, asynchronous replication does not enforce consistency between the replicated databases.

C. Snapshot Synchronization

In this model a snapshot1 is initially copied to all devices that require a replica of the database. From thereon, data is refreshed periodically. This can be a long process as database snapshots can be large in size and can take a lot of resources, thus this model is not the most suitable one for mobile devices. Usually, this model is used in read-only data distribution settings, where data is not updated frequently.

D. Push and Pull Data Synchronization

Push and pull data replication model includes snapshot synchronization and near-real time synchronization scheme. A near-real time synchronization scheme employs triggers at each local database which start the synchronization process each time a database is updated in order to propagate changes to other nodes. The difference between push and pull data is who triggers the process. In the push strategy, the device where modifications are made controls the process. In contrast, in the pull data strategy, a device polls if other devices have made changes. These schemes are preferred because they are less disruptive [18] while snapshot data replication is best suited when stale data can be tolerated [23].

2.2.3. Synchronization protocols

A synchronization protocol defines the steps and the workflow of the synchronization process in order to add, delete, or replace data items in devices. The main task of a data synchronization protocol is to “quickly identify changes, resolve possible conflicts, and propagate updates to the various synchronizing devices” [14].

Synchronization protocols for mobile devices should be carefully designed because of the limited computational resources of the devices and the unstable and inconsistent performance nature of network connections. After testing different protocols in mobile devices, Agarwal claims that most protocols often do not scale well with the number of devices involved in the process and with the size of the data to be synchronized [14]. In this sections, single device as well as multi devices synchronization protocols are analysed.

A. Single device synchronization protocols

When data can be modified in only one device, single-device synchronization protocols can be used. The mobile devices synchronize data to a central repository by maintaining modification flags or timestamps. By using this method, mobile devices and the central repository keep track of the data items that have been added, modified, or deleted since their last synchronization. Conflicts arise when the same data item has different versions on both the central repository and the mobile device, the choice of which modification to keep can be determined manually by the user or using different algorithms depending on the nature of the information to be synchronized. For single device synchronization in mobile environments, two main protocols can be used according to Agarwal [14]: Fast Sync Protocol and Slow Sync protocol.

1 Snapshot is a complete copy of a database in a single point of time
i. Fast Sync Protocol

The Fast Sync protocol maintains status flags in each data item to determine what changes have occurred since the last synchronization. These flags are updated whenever a data item is inserted, deleted, or modified. When the synchronization process is triggered by a mobile device, all data items whose flags have been changed are sent to the central repository. The central repository compares these data items with the ones it keeps in its database and one of following actions can be taken:

- The data item is created in the central repository if it was not available before.
- The device data item replaces the central repository version.
- The data item is deleted in the central repository.
- The data item is archived in the central repository if it was deleted in the mobile device.

Finally, the central repository sends its corrections back to the mobile device and all flags are reset. This protocol cannot be used when two or more devices need to synchronize with each other because flags should be reset in each synchronization process even if they are needed for synchronizing a second device.

ii. Slow Sync Protocol

The second protocol, Slow Sync is suitable when only a backup of the data is needed. This is because it makes a copy of the whole database in each synchronization. As well as Fast Sync, this cannot be used in a multi-mobile device environment. Additionally, this protocol has a big disadvantage which is that that the duration of the process increases linearly with the number of records stored in the devices [15]. Consequently, this protocol becomes unsuitable for using it in mobile networks where availability and stability is not guaranteed and bandwidth has a cost.

B. Multi-device synchronization protocols

Implementing synchronization among multiple mobile devices is considerably more difficult, because maintaining data consistency across several devices might require many individual synchronizations between pairs of devices [14]. This problem is discussed by different authors [4],[21],[14],[19],[20], who provide different solutions, each of them with different advantages and drawbacks. Two solutions that are widely cited in academic literature will be discussed in the following paragraphs. One of them is SyncML, which is an open industry initiative supported by hundreds of companies and which seeks to provide an open standard for data synchronization across different platforms and devices [24]. The second solution is the synchronization protocol proposed by Mascolo et al. at the journal article XMIDDLE: A Data-Sharing Middleware for Mobile Computing [25].

i. SyncML

The SyncML standard assumes that each device maintains modification flags for each of its data items with respect to every other device on the network in order to minimize communication time. In contrast with Fast Sync where modifying a record on a device toggles one set of status flags, in SyncML one set of status flags is toggled for every device on the network. After finishing the synchronization only the flags corresponding to these devices are reset [24]. Also, as a data item can be manipulated in multiple devices, it should maintain a set of modification information for each device in the network. This is why Agarwal [14] claims that the amount of memory needed to maintain this information can easily become a significant problem. It should also be noted that because it is a timestamp protocol, adding or removing a device from the network entails an update to every other device in the network.

ii. XMIDDLE

Mascolo et al. claim that XMIDDLE “provides an approach to sharing data that allows on-line collaboration, of line data manipulation, synchronization and application dependant reconciliation” among mobile devices and a central repository, and also directly among mobile devices [25]. In this approach, it is
assumed that each device stores its data in a tree structure in order to allow complex manipulation of data items. Tree structures allow to manipulate data items in different node levels, also provides hierarchy among nodes and basic relationships among the different elements. Therefore, XMIDDLE provides a set of primitives for tree manipulation which applications can use to access and modify the data.

In the XMIDDLE approach, a set of possible access points for the owned data tree is defined so that other devices can link to these nodes to get access to this data. A device needs to explicitly link to another device’s exported data branches in order to share data, but the link operation is not enough for data sharing among mobile devices. In order to share data, devices need to be connected. Devices may explicitly disconnect from other devices using a disconnect primitive, even though the connection among them has been not lost. XMIDDLE supports explicit disconnection to enable, for instance, a device to save battery power, to perform changes in isolation from other devices and to not receive updates that other devices broadcast. Disconnection may also occur due to network faults. In both cases, the disconnected host retains replicas of the last version of the trees it was sharing with other hosts while connected, and continues to be able to access and modify data.

The use of tree structures to organize allows to maintain the relationship among data nodes but it also raises a problem. Most applications use relational databases to store data and not XML tree structures such as the one proposed by Mascolo. This leads to the necessity to provide developers with a way to use the advantages of using a database and also a way to synchronize data while keeping data consistency inside the database scheme.

### 2.2.4. Conclusion

Data synchronization is difficult to implement. Multiple solutions have been provided which have advantages and drawbacks, but challenges to be faced are almost the same.

An important factor during synchronization is the amount of data that will be exchanged during the process. Incremental synchronization just exchanges the data items that have been modified on all systems since the last synchronization happened. Meanwhile, full synchronization exchanges all data items even if they were not changed. Popular synchronization methods use metadata such as timestamps and or flags to know the last modification and the synchronization state of a data item. Timestamps introduces the need to synchronize the local time of all participating devices which is difficult to achieve in mobile environments. An alternative to that approach is to compare and exchange data structures within data items, which is mostly done when synchronizing files. It makes necessary to transform non-structural data items into data structures that are equivalent.

Additionally, in distributed systems where the same data can be manipulated in different devices independently, several versions of data items can be generated. This data items should be merged into a consistent state between versions, but also a consistent state should be maintained for the possible references among these items [15] [26] [25].

In relational databases references are represented by foreign and primary keys. Maintaining consistency among these references implies a difficult challenge to face, making the use of concepts like transactions and referential consistency necessary [15]. All the synchronization solutions that have been detailed in the last section provide different ways to synchronize data items among devices. Nevertheless, most of these solutions depend on the data schema to be implemented. XMIDDLE uses a tree structure to keep this consistency, but this approach leads the developer to use a completely different data structure from the ones provided by Android, especially for the use of a relational database like SQLite. This problems raises the need for looking for different solutions.

The last issue leads to the search of new ways of storing data where the data model can be more flexible than in relational databases. The relational data model is based on defined relationships between tables and a strict column structure, all of which are explicitly organized in a database schema. In contrast, a
NoSQL database is able to accept structured, semi-structured, and unstructured data, providing a different way to store data items more easily and how data items are understood as synchronization data units.

2.3. SQL Databases and NoSQL Databases

The relational database model has dominated the database world for the last decades since it was introduced by Edgar Codd in 1969 [27], but in the last decade a new kind of database model has emerged. NoSQL databases present a new alternative based on non-relational models. In this section the characteristics of both, relational and non-relational databases will be analysed and compared in order to show the advantages and drawbacks that each model presents for this project. First, a brief summary about well-known SQL databases will be presented in order to have a basis to compare them to NoSQL Databases. Then the rationale for choosing a NoSQL option will be presented.

2.3.1. Relational Databases and SQL

In relational databases it is necessary to define a schema before working with data. This schema models the ways data is stored in the database. It presents a decomposition of data into tables consisting of rows and columns. Each row of the table is composed of cells that contain scalar non-structured data. Each of these cells are contained in a column that represents an attribute containing the same type of data. Each row usually has a key which is used to uniquely identify each row, to order the rows inside the containing table and to map data among table relationships [28].

This model is able to store data with minimal redundancy while guarantying data integrity. Codd defines two types of integrity: entity and referential integrity [27]. The former says that a table should contain a unique value for each row, while the latter focuses on the relationship between tables, where a foreign key must correspond to an existing row in the parent table.

SQL (Structured Query Language) is used to perform operations over stored data in relational databases. SQL allows applications to perform the basic task of creating, reading, updating and deleting data, also known as CRUD operations. SQL also supports indexing mechanism, which benefits a fast data access avoiding full table scans when queries comprises multiple tables. Another important concept in SQL are joins, which are read operations that combine multiple tables in order to return the data contained in them as they were in only one table [29].

After this brief introduction to relational model, the transaction model in relational databases should be particularly analysed. This topic will be detailed in the following paragraphs.

A. Transaction and ACID Properties

The relational model was conceived based on a centralized system with support of ACID transactions where relationships between tables are a core functionality together with protection mechanisms to keep the consistent state of the database. This conceptualization is opposite to the one that rules NoSQL databases which are conceived as distributed systems that provide eventual consistency, as it will be explained latter.

A transaction can be defined as a sequence of operations over data that must be either executed completely or totally rejected. This allows to ensure that states that satisfy all integrity conditions are persisted in the database. If a sequence of operations that constitute a transaction are executed partially, an inconsistency will occur. This could happen because of failures of the system or network disconnections. This is especially important in mobile environments where the network connection and other factors could cause interruptions in the operations.

---

1 This argument could be not totally true nowadays but it is for the case of SQLite.
As introduced before, an important characteristic of relational databases is that they follow ACID properties to ensure data reliability. Database theory defines 4 consistency properties (ACID properties) [30]:

- Atomicity: A transaction must be executed as a whole or not at all.
- Consistency: Only consistent states must be stored in the database.
- Isolation: Transactions must not be affected by other concurrent transactions.
- Durability: Once the client has been informed about the successful finalization of a transaction, the transaction must be stored persistently.

These rules are supported by different mechanisms in a database management system. Common mechanisms are two phase and rollback mechanisms, logging of operations and locking of data. This not only causes processing overhead but also creates challenges for databases synchronization in distributed and mobile environments. This leads us to an important question, whether it is possible to keep consistency, availability and tolerance at the same time in a distributed database system. This question will be answered in the next section using the CAP theorem.

### B. Consistency, Availability and Partition Tolerance in Databases

Brewer [31] proposed the CAP theorem which states that any network-shared data system can guarantee at most two out of the three of the following properties: consistency, availability and partition tolerance. In relational databases the emphasis is made in consistency and availability over data distribution over networks. This allows to persist data in a fast and reliable way supporting they scalability in the development of hardware performance. Some alternatives in choosing these properties are illustrated in Table 2-1.

<table>
<thead>
<tr>
<th>Choice</th>
<th>Traits</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency + Availability</td>
<td>2-phase-commit cache-validation</td>
<td>single-site databases,</td>
</tr>
<tr>
<td>(Forfeit Partitions)</td>
<td>protocols</td>
<td>cluster databases,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xFS file system</td>
</tr>
<tr>
<td>Consistency + Partition</td>
<td>pessimistic locking</td>
<td>distributed databases,</td>
</tr>
<tr>
<td>tolerance (Forfeit Availability)</td>
<td>make minority partitions unavailable</td>
<td>distributed locking</td>
</tr>
<tr>
<td>Availability + Partition</td>
<td>leases</td>
<td>Majority protocols</td>
</tr>
<tr>
<td>tolerance (Forfeit Consistency)</td>
<td>conflict resolution optimistic</td>
<td>Coda</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Web caching</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DNS</td>
</tr>
</tbody>
</table>

Table 2-1. CAP Theorem Alternatives and traits.[31]

With the expansion of the internet, the need of managing big, flexible and distributed data has emerged. This can be appreciated easily in mobile applications where data must be replicated in different devices and should be always available, even at the expense of data consistency.

### C. Conclusions

The goal of this project is to provide Android developers an API that enable users to create and modify data in the absence of a network connection and then synchronize these changes when the device’s network connection is restored. Therefore, the proposed system should be partition tolerant and should provide data availability.

SQLite, as most relational databases, is designed to guarantee ACID transactions. This creates a big challenge because ACID transactions depend on the nature of the data that is being manipulated and relational databases are designed to prioritize consistency over distribution tolerance. Also durability, one of the ACID properties, is difficult to achieve in environments where simultaneous modifications can occur in different devices.

In most mobile applications the data contained in the device is not very complex, no complex queries are performed over it and stale and temporal inconsistencies in data can be tolerated. This leads to 2 options,
keep acid properties at transaction level or at row level. To perform data synchronization at transaction levels it is important to know the nature of the data to be manipulated, and in this case this is not possible because one of the goals of this project is to generate a solution that can be used with a wide variety of schemas. On the other hand, performing synchronization at row level does make it difficult to keep relational consistency.

This conclusion makes it imperative to look for a new way to synchronize data, and a path that can be taken is to perform synchronization at an intermediate level, where rows can contain complex types of data and can be managed as isolated data items. Of course the data items cannot be totally isolated from one another. This implicates the existence of conflicts and multiple versions of the same data across different devices. This leads our research to NoSQL Databases, which are conceived for being used in distributed systems.

2.3.2. NoSQL Databases

The NoSQL term (Not Only SQL) not only refers to the use of the Structured Query Language to perform operation in database bases, this term is usually used to refer to non-relational, non-ACID and schema flexible databases [32]. This term was conceived to describe non-relational databases that emerged in the last years.

The NoSQL movement started as an effort to manage the big data phenomenon which arises with the web 2.0 era in situations where the use of relational databases is not feasible. Google and Amazon where pioneers of this movement developing customized databases to fulfil their particular requirements. Common characteristics of these databases are schema flexibility, eventual consistency support, non-ACID databases, capability to manage huge amount of data and the most important factor for this project, easy replication and synchronization support.

Nowadays there are more than 150 different databases catalogued as NoSQL [33]. This wide variety makes it difficult to make a decision choosing NoSQL Databases, but according to Kaur [32], they can be catalogued in 4 different categories: Column Oriented, Key Value, Document and Graph Databases. Before analysing each of these categories some common characteristics of NoSQL databases will be presented.

A. NoSQL Adoption Drivers

In the following paragraphs some of the main characteristics of NoSQL Databases are discussed, emphasizing on the ones that are relevant for this project:

- **Schema Flexibility**: In contrast with relational databases, NoSQL databases are showcased as schema-less [32] databases and according to Couchbase[34], this is one of the main reasons why businesses are adopting this kind of solutions. Even though this is one of the most popular characteristics of these databases, some authors think that this is not totally true [4], [32], [34]–[36]. Most of them think that the schema is implicit and it must be defined by the application because it should parse data when retrieved from the database. In case the schema is inconsistent the application itself must deal with this, in contrast with relational databases where the database manager takes care of this issue.

  This means that the database does not control if different documents have the same schema, and that data can be partially updated without data loss as it could happen in databases where data follow a strict schema. At the moment of data synchronization this characteristic is very important because it allows to manage conflicts in an easier way if the process fails or is interrupted.

- **BASE approach.** In contrast with ACID approach of relational databases, the BASE approach adopted by NoSQL databases forfeits consistency and isolation in “favour of availability, graceful degradation and performance” [31]. A comparison of both approaches was made by Brewen [31] which is summarized in Table 2-2.
<table>
<thead>
<tr>
<th>ACID</th>
<th>BASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Strong consistency</td>
<td>• Weak consistency – stale data tolerance</td>
</tr>
<tr>
<td>• Isolation</td>
<td>• Availability first</td>
</tr>
<tr>
<td>• Focus on “commit”</td>
<td>• Best effort</td>
</tr>
<tr>
<td>• Nested transactions</td>
<td>• Approximate answers tolerance</td>
</tr>
<tr>
<td>• Availability</td>
<td>• Aggressive (optimistic)</td>
</tr>
<tr>
<td>• Conservative (pessimistic)</td>
<td>• Simpler!</td>
</tr>
<tr>
<td>• Difficult evolution (e.g. schema)</td>
<td>• Faster</td>
</tr>
<tr>
<td></td>
<td>• Easier evolution</td>
</tr>
</tbody>
</table>

Table 2-2. ACID vs BASE.[37]

- **Eventual consistency**: the first property is closely related to this. Due to the distributed nature of this kind of databases they cannot guarantee strict consistency. This can result in inconsistent state of data while updates are being performed [37]. This could be a weakness for applications where consistency is essential (for example financial applications) but this can be a great advantage in applications where not all read operations must retrieve the same information to all users because replications in different nodes can be made more easily than in relational databases.

- **Application Development Agility**: In relational databases the schema should be changed before application when data requirements evolve. This limits the way application features can be added and updated. In contrast in NoSQL databases, the “focus shifts to domain design” [38] promoting agile development and evolutions of applications. This is especially important for mobile applications which evolve fast and in distributed environments where the change of database schemas creates new challenges at the moment of distributing new applications.

- **Non-Relational**: This property is an advantage when managed data has simple structures or its structure is flexible as it usually happens with data managed by mobile applications. To illustrate this, Mai [39] introduces the example illustrated in Figure 2-1 and Figure 2-2. In SQL databases the student_course table repeats the field Std_ID several times for each course, while in a NoSQL approach the database is flexible enough to map a student with a list of courses in only one record. This also implicates that keeping consistency during data synchronization will be easier because relational consistency is not as strict as in relational databases. In other words it prioritizes availability over consistency in NoSQL databases.

- **Performance and Scalability**: In contrast with relational databases, NoSQL databases are designed to scale well horizontally, not relying only on hardware capacity [39]. Clusters can be added without stopping the service providing better availability and distributed parallel processing which increases performance. This is especially important in mobile applications where the numbers of users and the amount of data that will be managed cannot be easily estimated because the number of users is usually unknown and can change rapidly.

![Figure 2-1. Relational model-Student example](image1.png)

![Figure 2-2. Non-relational model-Student example](image2.png)
- **Map Reduce Model and Queries**: Relational database are the best choice for performing complex queries in complex data structures but for a large amount of data this can lead to delays for responses. In contrast, NoSQL databases usually do not provide support for complex queries but deliver better performance in large amounts of data using the map reduce model. In common mobile applications complex queries are usually not performed and the amount of data managed in mobile devices is usually small. On the other hand the ability to perform fast queries in big amounts of data is useful in a central repository where all the data of all users is kept together. Also, the fact that NoSQL databases distributions usually use different languages or APIs for querying data should be considered, which creates the necessity of learning the particular characteristics of each distribution when one of this databases is adopted. On the other hand, most developers are familiar with SQL, which is used in relational databases like SQLite in Android.

i. **Conclusion**

After going through the main characteristics of SQL and NoSQL databases a valid conclusion is that they target different domains. Lamllari [38] provides a classification of databases according to the CAP theorem that could be seen in Figure 2-3. This leads to the conclusion that when consistency and complex queries are necessary, the best choice is relational databases. In contrast, when availability, scalability and flexibility are more important, the best choice are NoSQL databases. Table 2-3 summarizes the most important differences between relational databases and NoSQL databases.

![Figure 2-3. CAP Theorem and Databases][38, p. 23]

In the case of most mobile applications, where eventual consistency is not a problem and availability and partition tolerance or replication are main goals for the developer, NoSQL databases are suitable to be used. Additionally, it is worth noticing that most developers are familiar with relational databases, which is almost a standard in industry, making the adoption of NoSQL databases more difficult.
<table>
<thead>
<tr>
<th>Data Model</th>
<th>Relational Databases</th>
<th>NoSQL Databases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure</td>
<td>Organized in a rigid schema which comprises tables, rows and relationships rigid relationships among tables.</td>
<td>Flexible schemas or schema with fewer data structures. The organization of data varies with the type of database.</td>
</tr>
<tr>
<td>Data Types</td>
<td>Strongly typed</td>
<td>Flexible</td>
</tr>
<tr>
<td>Transactional model</td>
<td>ACID</td>
<td>BASE (Schema flexible, ability to manage unstructured data, supports fast data and application evolution)</td>
</tr>
<tr>
<td>Adoption Driver</td>
<td>Consistency</td>
<td>Availability and distribution</td>
</tr>
<tr>
<td>Query</td>
<td>Uses widely known SQL which supports complex and ad-hoc queries.</td>
<td>Different APIs depending on the distribution (REST, client libraries, protocol buffers). Most do not support complex and ad-hoc queries easily.</td>
</tr>
<tr>
<td>CAP classification</td>
<td>Consistency-availability</td>
<td>Availability – distribution tolerance/ or consistency-distribution tolerance</td>
</tr>
</tbody>
</table>

**Table 2.3. SQL Databases Vs NoSQL**

Furthermore, as NoSQL databases are suitable for this project, the next section will analyse the different types of non-relational databases that can be used.

### B. Types of NoSQL Databases

Several taxonomies have been proposed for the classification of NoSQL Databases [32], [37]. This section is based in Kaur’s [32] classification where 4 types are proposed, column oriented, key value, document and graph based databases. Special emphasis will be done in document databases and why they are the most common model adopted in mobile environments [10], [34], [38], [40] [41].

#### i. Column Oriented Databases

Column databases are also referred as Big Table clones because they are based on Google’s Big Table model [32], [38]. The main goal of this category is to provide the capability of managing large amounts of data. They organize the information in key value pairs organized in a semi-schematized and hierarchical-column oriented pattern [42]. Figure 2-1 illustrates the data model used in the column oriented pattern. This pattern offers great performance in aggregation operations and ad-hoc and dynamic queries in big amounts of data. Several authors [32], [38], [42] conclude that these categories of databases are best suited for analytical purposes.

![Figure 2-4. Column Oriented Databases Data Model](43)
ii. Graph Databases

This type of database models data as a network structure which contains nodes related by edges. Both, nodes and edges can contain data and relationships are represented by edges which can be directed or not. This model is illustrated in Figure 2-5.

![Figure 2-5. Graph Database Data Model. [43]](image)

This kind of databases is optimized for representing data as graphs structures which makes them suitable for finding relationships in big amounts of data. Some authors [32], [44] claim that this kind of database is most used in social networking sites where relationships between data are as important as data itself. The graph data structures are not the data model targeted by this project which makes this type of database not the best choice.

iii. Key Value Databases

This kind of databases is based on a simple model where data is organized as “an associative array of entries consisting of key-value pairs” [32]. Each key is unique and is used to get back the data stored in the values associated with it. Values contain only scalar values and cannot embed an object inside another object. Kaur argues that this kind of databases can be seen “as relational databases having multiple rows and only two columns: key and value” [32]. This model is illustrated in Figure 2-6.

![Figure 2-6. Key Value Databases Data Model. [43]](image)
This kind of databases is suitable when the schema of data changes frequently and where eventual consistency can be tolerated. A drawback of key-valley databases is that they do not provide rich ad-hoc and analytical queries. This kind of databases can be applied in this project because they promote an agile application evolution and distributed databases. The lack of complex queries is not a deal breaker as most mobile applications do not use them.

iv. Document Databases

This kind of databases can be seen as an extension of the key value model where the value can be not only scalar values, but also another document or an array of documents or scalar values. Documents can be addressed by their keys which should be unique. This data model is illustrated in Figure 2-7.

In some databases a foreign key mechanism is implemented which allows referencing among documents. Despite this referencing mechanism, most implementations lack basic referential integrity features like foreign key constrains. Therefore, the responsibility of keeping data consistency relies on the programmer [45, p. 22].

This data model allows to persist documents which have different structures with any number of fields inside them. This also promotes the evolution of applications as well as fields and structure of documents can be easily changed [45].

Another important feature that is worth to be pointed is that this kind of databases provides rest to APIs which can be used to retrieve documents and to replicate and synchronize databases. A good example of these features can be found in CouchDB and Couchbase databases [46][34].

Figure 2-7. Document Databases Data Model

v. Conclusion

In relational databases, an operation over data usually requires the manipulation of rows located in several interrelated tables making difficult to keep consistency during data synchronization or database replication operations. This makes NoSQL databases an interesting choice for this project as they support eventual consistency allowing data synchronizations in environments where failures are expected. This is especially important for mobile applications where network connectivity cannot be guaranteed.
Table 2-4 presents a comparison of the types of databases presented here. This was elaborated using Scofield’s ideas [47] and those by different researches regarding the use of databases in mobile environments [26], [32], [38], [40], [41], [41], [45].

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Scalability</th>
<th>Flexibility</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Value stores</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>Column stores</td>
<td>high</td>
<td>high</td>
<td>moderate</td>
<td>low</td>
</tr>
<tr>
<td>Document stores</td>
<td>high</td>
<td>variable (high)</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Graph databases</td>
<td>variable</td>
<td>variable</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Relational databases</td>
<td>variable</td>
<td>variable</td>
<td>low</td>
<td>moderate</td>
</tr>
</tbody>
</table>

Table 2-4. NoSQL Databases Comparison

There are several reasons why document databases are an attractive choice for this project including replication features, cloud synchronization services, and the same document data model which provides great flexibility to developers and low complexity. Nevertheless, the most important feature for this project is that synchronization can be achieved keeping data consistency if references among them can be minimized. This allows to provide a synchronization mechanism that can be useful in different cases and that is low coupled with the structure of the manipulated data.

Despite their powerful nature, documents databases also have drawbacks. For example, in order to retrieve a value of a record inside a document, the whole document should be retrieved, and the same issue exists for updates, affecting performance.

Additionally, most NoSQL databases are designed to work in devices with high computational power and trustable network infrastructure. Perrier [41] presents a recent work where he evaluates the performance of NoSQL based embedded databases on Android devices concluding that they performed not as well as SQLite. It is clear that the incursion of this kind of databases in the mobile world is recent, so there is not a wide variety of options to choose from.

This project needs a database that could be distributed together with the solution under development. The features that this database should ideally provide are:

- Based on the document data model.
- Implemented as an Android native database engine that could be embedded in applications.
- Distributed under a free software license.
- Supported by a wide community and documentation should be available.
- Ideally, have a server side with which can easily interact without needing data manipulation.
- Ideally, provide data synchronization or database replication features.

The database that best accomplishes this goals is Couchbase database which will be analysed in the next section.

Another possible way is to create a “polyglot” [48] solution where different data models are used in the device and in the central repository. This solution comes with a high cost in complexity. Each database will introduce a different interface to be learned and the conversion between data models is a complex task that could dramatically increase the complexity of this project.

2.3.3. Couchbase and Couchlite

A. Why Couchbase and Couchbase Lite

After searching for documents databases that could be distributed together with Android applications, the conclusion is that things have not changed since Perrier [41] argued that little work has been done around embedded NoSQL databases in Android.
Some attempts to port CouchDB to Android can be found in the internet. Most of them done by enthusiasts of the NoSQL movement rather than consolidated communities or companies [49][50][51]. Other projects like CouchIO and TouchDB have been abandoned to be consolidated in the project Couchbase Mobile [52][53]. Even on the wiki web page of CouchDB, where the instructions for installing it on Android are located, the first statement says that those instructions are out of date, together with a reference to the Couch Mobile project [54].

An option that is worth to be mentioned is Cloudant. This company offers a solution where data is stored in a local database in mobile devices and provides synchronization libraries to replicate this information in a remote central repository that could be based on a Cloudant or a CouchDB database. Nevertheless, this company does not provide documentation and a stable version for this and the Android version is still under development [55], [56].

The ideas exposed in the previous lines support the claim of Couchbase that says that “Couchbase Mobile is the first and only native NoSQL JSON mobile database solution” [57]. Additionally Couchbase Mobile is not only the Couchbase Lite engine database for Android, it also includes two other components: Couchbase Sync Gateway and the NoSQL database Couchbase server. Couchbase Sync Gateway is a synchronization tier for data replication while the server is a document-based database. The main features of this set of tools are summarized in Figure 2-8.

![Figure 2-8. Couchbase Mobile Features. (57)](image)

It must be pointed out that this is an open source project distributed under the Apache 2.0 License. An enterprise version is also available if support is needed, but support can also be obtained from the open community that promotes this project [58][59]. It should also be noted that different resources are available to learn about this project, such as documentation, blogs, webinars, papers and others [60].

After this introduction, the main features of this product will be exposed and how these can support the development of this project.

A. Couchbase Mobile Architecture

As it was mentioned before, Couchbase Mobile comprises 3 main modules: Couchbase Lite, Sync Gateway and Couchbase server. Couchbase Lite is a document database engine that provides tools for storing documents in JSON format and for data replication. Sync Gateway provides a framework to replicate information that is created or manipulated in mobile devices to a central data repository. The central repository was implemented using the Couchbase server. Figure 2-9 provides a general view of the components that conform Couchbase Mobile and which will be detailed in the following subsections.
i. Couchbase Lite

Couchbase Lite is a NoSQL document based database engine that can be distributed together with Android applications in an embedded way. It is distributed together with an Android API, which allows developers to perform basic CRUD operations over data. This database can work as an independent local database in mobile devices, but it also can act as a facilitator for peer to peer replication along with Sync Gateway and Couchbase server. The most important features of Couchbase Lite that benefit this project are summarized in Table 2-5.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefits</th>
</tr>
</thead>
</table>
| Document oriented and flexible schema | • Stores data in JSON format with a flexible schema model in contrast with relational databases.  
  • Binary data can be attached to documents, such as multimedia content. |
| Document index, querying, and sync capabilities | • A map reduction model is used to support filtering and querying data stores in the database.  
  • Supports creation, indexes and views thus providing flexibility in queries. |
| Filtering and replication features | • Provides facilities for one shot or continuous replication.  
  • Supports data filtering to limit which data is replicated and which is not. |
| Conflict resolution features      | • To manage conflicts, this databases uses multi version concurrency control.  
  • For each document a set of versions can be retrieved from the database in order to resolve conflict if it is needed. |

Table 2-5. Couchbase Lite Features. [60][57]

ii. Couchbase Sync Gateway

Sync Gateway server manages the interchange of data between mobile devices and the central database. It provides support for control data access and routing so that all information of multiple users stored in multiple devices can be managed from a central database.

Data is routing using “channels” in order to define which portions of data should be replicated in the central database server and in mobile devices. Channels also provide access to information to users limiting the portion of data that is available for each of them and allowing users to share information between them.
The most important features of Sync Gateway that provides benefits to this project are summarized in Table 2-6.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronization API</td>
<td>• Capability easily to configure access and routing of data.</td>
</tr>
<tr>
<td></td>
<td>• Abstraction of network layer.</td>
</tr>
<tr>
<td>Custom data flow</td>
<td>• Support of multi master data replication to and from the central server and mobile devices.</td>
</tr>
<tr>
<td></td>
<td>• Support of uni-directional and bi-directional data flow to and from the central server and mobile devices.</td>
</tr>
<tr>
<td>Control of data routing, authentication and access control</td>
<td>• Support for configuration of document access, authentication, and data routing.</td>
</tr>
<tr>
<td></td>
<td>• Prebuilt authentication features</td>
</tr>
<tr>
<td>Scalable</td>
<td>• Because of the stateless nature of Sync Gateway, an additional server can be added to support a big amount of requests.</td>
</tr>
</tbody>
</table>

Table 2-6. Sync Gateway features. [52][49]

iii. Couchbase server

Couchbase server is a NoSQL database that offers a flexible data model based on documents. It is easily scalable and provides high performance and availability for managing big amounts of data. As the nature of this kind of databases already was discussed in section 2.3.2, the arguments in this section will be based on reasons why Couchbase server was selected over CouchDB even though both products are supported by Couchbase Mobile.

CouchDB and Couchbase have several things in common. Both of them are NoSQL databases based on the document model and both use JSON as their document format. Also indexing and querying approaches are very similar because CouchDB code was used in Couchbase development in an initial stage [62]. Also replication technology of CouchDB is the “underlying technology for Couchbase datacentre replication” [62]. Finally, one of the most important similarities between both is that they are totally open source and licensed under the Apache 2.0 licence.

Even though CouchDB and Couchbase projects are closely related they have substantial differences. Couchbase has a built in memcached-based technology which provides better performance in writing and reading operations while CouchDB is a disk-based database. Additionally, CouchDB provides a very useful HTTP API which supports data manipulation and querying. Finally, even though Sync Gateway project claims to be compatible with both servers, it is supported and documented for being used with Couchbase only. This final fact is one of the most important reasons why Couchbase was selected over CouchDB.

iv. Conclusions

Data managed in mobile applications has evolved. Not only simple data based on text and numbers is offered to users in mobile applications, now mobile applications incorporate rich multimedia contents and complex data which is constantly changing. Additionally, the amount of data that mobile users create have rocketed during the last years. It can be said that relational databases are the standard for Android but are limited to face these new challenges. This makes the adoption of NoSQL an attractive choice.

The stack of technologies provided with Couchbase Mobile gives developers the ability to use flexible data structures and abstract data synchronization. This makes easier to develop applications which support offline creation and manipulation of data, and then synchronize this data in central repositories. This group of technologies also provides tools for authentication, network communications, data access control and data routing control thus allowing this project to provide a more robust solution in a faster way.
Of course the adoption of these tools entails some drawbacks. The most important is that developers of Android applications are not usually familiar with NoSQL Databases, which creates a new challenge for this project: provide an API that allows developers to create applications without having to learn how to use this kind of databases. This API should not only hide from developers difficulties that synchronization and NoSQL database rises, but also provide a general solution for multiple data schemas that different application could need. Model Driven Software Development provides facilities that can be used to create this API, creation customized solution for different requirements. This topic is analysed in the following chapter.

2.4. Model Driven Software Development (MDSD)

MDSD bases the development of software on models. From a model, an automatic transformation can be applied to generate source code for different applications in different platforms [63], [64]. In the case of this project, MDSD provides tools that allows to generate the proposed API from a data model generated by the developer according to the specific requirements of each project. Additionally, the generated system should run in two different platforms, one for the server side and another for the mobile application [65], [66]. This makes MDSD especially useful because an Android developer can avoid the need of learning how to develop in different platforms. A unique model can capture the data requirements of the mobile application on a comparatively high level and this model can be translated into code to produce an API to manage data in the mobile application and the necessary logic to synchronize it in the server side.

Heitkötter and Majchrzak [63] define a set of requirements that are common for data driven applications. One of them is to “define data types, and access, create, retrieve, update, and delete operations (CRUD) for these types on the device” [63] which is similar to the aim of this project. Their solution, MD2, prove that MDSD is a suitable solution for the generation of data driven mobile applications.

One of the dominant approaches of MDSD is Model Driven Architecture [67]. In the next subsections some concepts of this approach will be discussed.

2.4.1. The Model Driven Architecture (MDA)

Model Driven Architecture (MDA) is a framework defined by the Object Management Group (OMG) for software development [68]. MDA allows developers to reduce developing time of software in a significant way using a set of concepts like meta-models, meta-data and models and focusing primarily on the functionality and behaviour of the applications rather than on the technology in which it will be implemented [67]. This allows to separate implementation details from business logic supporting the main aims of MDA: portability, interoperability and reusability [64], [67].

MDA also defines a development lifecycle which is illustrated in Figure 2-10. Even though this is very similar to traditional deployment life cycles, the principal difference is the nature of artefacts that are created during the process: they are formal models and transformations of this models [65].

In MDA model are categorized in three abstraction levels [65],[67], [68], [68]:

- Computation Independent Model (CIM): This model defines the system requirements in a high level of abstraction, independently of the technologies that will be used to implement the system.
- Platform Independent Model (PIM): This is a refinement of CIM and focuses on the operation of the system defining the logic of the system and its interactions with other systems. This model abstracts implementation platforms.
- Platform Specific Model (PSM): This model presents a view of the system for a specific platform. Here are combined the specification made in PIM with the details needed for the use of the specified platform.
2.4.2. Conclusions

The main motivations for MDSD is to improve productivity, consistency and quality software products [69]. This project’s aim is to use MDSD principles to provide developers with a tool that generates an API that allows Android applications to synchronize data with a central repository. A MDSD approach is promising because MDSD based tools allow developers to improve their productivity and provide better functionality [69]. Additionally, by abstracting out implementation details of data management and synchronization, the tools provided with this project will reduce implementation complexity allowing developers to focus on other details thus improving the quality of their applications.

2.5. Domain Specific languages

In order to abstract out implementation details from developer, the system should provide a way to model the data requirements of the application under development where data management and data synchronization concepts can be defined. Then, this concepts can be mapped to an implementation by the code generator to generate the API that the developer will use.

Before discussing about domain specific languages (DSL) it is necessary to provide a clear definition of them. Van Deursen et al provide a clear definition for domain specific language:” A DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain” [70]. This definition says that a key property of DSL is their focussed expressive power and this property is what makes it a concept to be considered in this project.

The use of a DSL approach in this project comes with advantages and drawbacks. The most relevant advantages for this project will now be presented:


- **DSLs** allows developers to express solutions using the notation of the problem domain and at the level of abstraction that is needed for this domain [61], [62]. This way the cost of dealing with technical details at stages when they are not needed is minimized.
- Better maintainability, better reliability and higher productivity can be achieved when the developer can work at a higher abstraction level [71], [72]. This also leads to a more portable and reusable software, ideal features for MDSD.
- **DSLs** keeps domain notation and knowledge which allows conservation and reuse of this knowledge [70], [73]. This also makes it easier to understand and reuse.

In contrast with these advantages, the creation of a DSL also comes with drawbacks. The most relevant will be presented below:

- The task of developing a new DSL specific for a domain is a complex task which involves high costs [72], [74]. Not only the costs of designing, implementing and maintaining a DSL should be considered but also the costs of training for DSL users.
- Some authors claims that software at high level of abstractions can lead to inefficiencies in the final product compared to hand coded software [70], [74].

### 2.5.1. Conclusions

The use of a DSL should be carefully considered. DSL development is hard and requires domain and language development knowledge. The high cost issues can be mitigated because they can be amortized over many products but should be noted that time is a limited resource in this project.

Other projects where the use of DSL languages for data modelling already has been tested should also be considered. Good examples of this are:

- **Pongo** – Java POJO generator for MongoDB [75].
- **MD2** - Cross-Platform Model-Driven Development of Mobile Applications with MD2 [63].
- **Model Driven Data Management in Android with the Android Content Provider** [76].

All these projects have one common characteristic: all of them uses the Eclipse Modelling Framework (EMF) and ECore to provide user a meta model which can be used to define the data schema. Ecore is the core meta model of EMF [77] and as a foundational meta-model can be used as a meta-model itself or can be used as a base to define domain specific meta-models. Additionally, it is worth to notice that Ecore is considered as “defacto” reference implementation of the Meta-Object Facility of the Object Management Group (OMG) standard for model driven engineering by some authors [77], [79]. The fact that Ecore is almost a standard in the industry and that most of the modelling concepts of Ecore are familiar to developers, especially for object oriented programmers; since it unifies Java, XML and UML makes its use a promising choice for this project.

Some of these projects also use the Emfatic editor, which is a feature rich text editor which can be used by the developer to create an ECore model in a textual form using the Emfatic language. The use of Emfatic language allows to reduce implementation overhead, proving a way to represent a model in a textual form using a simple language in a feature-rich text editor extending the features provided by Ecore.

Ecore and Emfatic can be used to create a declarative language in the sense that developer specifies what the application should accomplish without caring about how to accomplish it algorithmically. Thus, he developer can focus on the problem that the application is solving.

---

1 OMG develop enterprise integration standards for a wide range of technologies and industries. OMG’s modelling standards includes the Unified Modelling Language (UML) and Model Driven Architecture (MDA). [78]
All these considerations lead us to think that the use of a widely used and extensively tested meta-model such as ECore in combination with a language such as Emfatic could provide a solution to the issues analysed in this section.

These lead this research to analyse how these tools can be integration in the Android development process and how then can be used by developer. This will be considered in the next section.

### 2.6. Integrated Development Environments (WTF, ADT and Epsilon)

One of the aims of this project is to create a tool that allows code generation for both, the server side of the system and for the mobile device. Before developing these tools it is necessary to consider some important factors like IDE (Integrated Developing Environments) support for the server development and for the generation of the mobile application. Available modelling technologies should also be considered and how these can be integrated to these IDEs in order to provide a friendly developing environment to the user.

Moreover, Android Application developing is usually made using Java and the server side should include a Java based database client, an IDE which supports developing in this programming language is ideal. The most widely used IDE for Java is Eclipse. Even Eclipse competitors as IntelliJ IDEA recognize this fact [80]. Eclipse provides wide support for the development of web services. A good example is the Eclipse Web Tools Platform (WTP). This set of tools extends the Eclipse platform for developing Web and Java EE applications [81]. This project includes source and graphical editors, wizards and built-in tools to simplify development, deploying, running, and testing web applications.

On the mobile application side, official Android developer’s documentation promotes the use of two IDEs: Eclipse and Android Studio [82]. The official Android developer’s web page distributes an Eclipse version called Eclipse ADT (Android Development Tools) which includes all necessary tools for Android development but ADT also can be integrated to other Eclipse versions. Steeg et al. [76] claim that this is the most popular platform for the development of native Android Applications. On the other hand, Android Studio is a new Android development environment which provides new features and improvements over Eclipse ADT, [83] but it is still under development and only a beta version is available. This makes Eclipse the better choice for this project from the Android development perspective.

Finally, Eclipse also provides concrete tools which support MDSD and MDA paradigms. A good example is the Eclipse Modelling Framework which “is a modelling framework and code generation facility for building tools and other applications based on a structured data model” [84]. Additionally, Eclipse also support tools like Epsilon (Extensible Platform of Integrated Languages for mOdell maNagement,) which is a model management platform that provides a collection of interoperable languages for performing tasks over models like model transformation, code generation, merging, refactoring and validation. These languages are platform-agnostic and therefore can work with any type of structured data model of a wide range of meta models and technologies [85], [86].

#### 2.6.1. Conclusions

As was pointed before, the aim of this project is to create a friendly tool that supports the creation of Android applications. A promising way to implement this is through an Eclipse plugin because this IDE supports the development of all the modules that this kind of system needs. The decisive factor for choosing this IDE are explained below:

- Eclipse architecture offers to developers a modular, simple to extend, widely documented and community supported solutions for plugin development.
- Eclipse ADT is the official tool for Android development.
- The Eclipse Modelling Framework contains the meta model Ecore [84] which allows to describe models. According to Merks, it is the “de facto reference implementation of OMG’s EMOF” [77]. Ecore as a foundational meta-model can be used as a meta-model itself or can be used as a base to define domain specific meta-models. Additionally, EMF provides support for querying models, code generation, standardised model serialization and model editors creation [84].
- Epsilon provides a set of languages for performing a wide variety of tasks over models such as validation (EVL), transformation (ETL), code generation (EGL), comparison (ECL), merging (EML) and migration (Flock). Additionally, Epsilon can be used with Ecore and EMF providing a wide range of possibilities for the development of a code generation tool [85], [86].

2.7. Summary
This chapter has reviewed the main concepts and technologies used in this project. Section 2.1 presents the advantages, disadvantages and common use cases for the different options that Android provides to persist data.

Section 2.2 presents main concepts of data synchronization and the challenges that developer should face to implement it. This section also presents different strategies and protocols used in several industrial and academic projects, analysing their benefits and weaknesses.

Section 2.3 offers a general view of databases, analysing their use in distributed systems. In this section relational and non-relational databases are compared in the context of distributed system concluding that NoSQL databases provides a suitable model for this project as well as they provides several advantages for data synchronization. Additionally, a review of the different types of NoSQL databases is done, where a detailed analysis of the document model is presented in the context of mobile devices. Finally, most important facilities that the Couchbase Lite framework provides for data synchronization in a mobile context is discussed.

Section 2.4 discuss Model Driven Software Development. This section also reviews How MDSD have been used in different projects for the development of mobile devices. Finally a brief review of one of the most dominant approaches of MDSD is analysed, the Model Driven Architecture approach.

Section 2.5 introduces main concepts of domain specific languages and how they have been used in related projects in the context of NoSQL databases and mobile applications.

Finally, section 2.6 discusses the most commonly used IDEs used for developing Android applications and how they can be integrated with modelling tools. A brief review of MDE tools, particularly the Epsilon framework is also presented here.
Chapter 3  METHODOLOGY

The benefits of using software development methodologies are widely recognized [87],[88], [89], particularly in the areas of project management, productivity, quality and communication. Most of these methodologies are based on the planning, analysis, design, coding, testing and delivery phases. These phases provide a guide to developers to define the different steps that should be followed during all the stages of the development process. In these sections, the rationale that led to choose a particular methodology will be explained.

Before choosing a methodology, some particular factors that will drive the selection of a methodology for this project should be considered:

<table>
<thead>
<tr>
<th>Factor</th>
<th>Criteria for this project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of team</td>
<td>1 person</td>
</tr>
<tr>
<td>Expectation of change</td>
<td>Requirements are not expected to change along the process. Nevertheless, because of the</td>
</tr>
<tr>
<td>requirements</td>
<td>lack of experience of the developer and the lack of real customers, it is difficult to</td>
</tr>
<tr>
<td></td>
<td>define all requirements at an initial stage.</td>
</tr>
<tr>
<td>Customer communication</td>
<td>The Project Supervisor assumes the role of end user.</td>
</tr>
<tr>
<td>Team roles</td>
<td>As the team comprises only 1 person, the developer should perform all the roles</td>
</tr>
<tr>
<td>Developing time</td>
<td>The time for the developing this project is short and cannot be negotiated.</td>
</tr>
<tr>
<td>Team experience in use of</td>
<td>The developer of this project must gain knowledge in most of the tools and technologies</td>
</tr>
<tr>
<td>technologies and tools.</td>
<td>that will be used. Technologies such as NoSQL Databases are totally new for the developer</td>
</tr>
<tr>
<td></td>
<td>and his experience in the areas of Android applications development, Model Driven</td>
</tr>
<tr>
<td></td>
<td>Development (MDE), and usage of MDE tools like EMF and Epsilon is very limited.</td>
</tr>
</tbody>
</table>

These criteria lead to answer the question of whether a plan-driven methodology or an agile methodology is the correct choice for this situation.

3.1. Plan-driven Methodologies and Agile Methodologies Comparison

Picek [90] proposes to classify modern methodologies into two groups: Plan Driven methodologies and Agile methodologies. Plan Driven methodologies emphasise detail planning, modelling and documentation while Agile methodologies emphasise that software should be created quickly, without redundant documentation, providing support for frequent requirement changes. The most important characteristics of both categories are illustrated in Table 3-2.

<table>
<thead>
<tr>
<th>Project characteristics</th>
<th>Plan-driven/traditional methodologies</th>
<th>Agile methodologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary goals</td>
<td>• Predictability</td>
<td>• Flexibility</td>
</tr>
<tr>
<td></td>
<td>• Stability</td>
<td>• Fast response to change</td>
</tr>
<tr>
<td></td>
<td>• High assurance</td>
<td>• Customer collaboration</td>
</tr>
<tr>
<td>Project Size</td>
<td>• Medium to large size projects</td>
<td>• Small to large size projects</td>
</tr>
<tr>
<td>Environment</td>
<td>• Stable</td>
<td>• Unstable</td>
</tr>
<tr>
<td></td>
<td>• Minimal changes expected</td>
<td>• Changes expected</td>
</tr>
<tr>
<td></td>
<td>• Focus on project and organization</td>
<td>• Focus on project</td>
</tr>
</tbody>
</table>
Customer relation

- As needed interactions
- Formally established in contract
- Dedicated onsite
- Informal interaction

Planning

- Documented plans
- Quantitative control
- Well-defined and strict activities
- Informal plans
- Qualitative control
- Flexible activities that are light for tracking

Requirements

- Formal requirements definitions
- Minimization of unexpected changes
- Informal stories and test cases
- Changes expected

Development

- Extensive design
- Longer increments
- Refactoring expected to be expensive
- Refactoring expected to be inexpensive
- Simple design
- Short increments

Testing

- Documented test plans
- Documented test procedures
- Executable test cases define testing and requirements

<table>
<thead>
<tr>
<th>Feature</th>
<th>RUP</th>
<th>RAD</th>
<th>FDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement specification</td>
<td>Beginning</td>
<td>Time-box released</td>
<td>Beginning</td>
</tr>
<tr>
<td>Understanding requirements</td>
<td>Difficult to understand</td>
<td>Easily understood</td>
<td>Well understood</td>
</tr>
<tr>
<td>Cost</td>
<td>Expensive</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Simplicity</td>
<td>Simple and clear</td>
<td>Very simple</td>
<td>Intermediate</td>
</tr>
</tbody>
</table>

Table 3-2. Plan Driven Methodologies and Agile Methodologies.

Agile methods are suitable for this project mainly because of the small size of the developer team, the limited time for developing process and the flexibility of artifacts that should be created during the process. Nevertheless, the problem complexity, the lack of experience of the developer in the problem domain and in the used technologies, as well as the impossibility of constant customer contact are better addressed by plan driven methodologies. Additionally, requirements are expected not to be unstable during the developing process, which is one of the main reasons to choose Agile methodologies. This reason leads to choose a plan driven methodology, but combining practices from both types of methodologies is beneficial for this project.

3.2. Plan-driven Methodology Selection

The next question to solve is which of the plan driven methodologies is more suitable for this project. The choice of the development methodology will be done among Plan Driven methodologies which support iterative development. This is because support for iterative and incremental development is one of the main characteristics needed for this project in order to address errors generated by the lack of experience of the developer.

Iterative and incremental development is supported by several methodologies [91]. Methodologies that were analysed for this project are: Rational Unified Process (RUP), Feature-Driven Development (FDD) and Rapid Application Development (RAD). After comparing methodologies which support iterative and incremental development, Rapid Application Development was chosen. The comparison between these methodologies is summarized in Table 3-3
<table>
<thead>
<tr>
<th>Risk evolved</th>
<th>Critical risks in the early stages</th>
<th>Very low</th>
<th>Varies according to level of complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expertise required</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Customer involvement</td>
<td>At the beginning and at the last face</td>
<td>Only at the beginning</td>
<td>At the beginning and at the last face</td>
</tr>
<tr>
<td>Time frame</td>
<td>Medium</td>
<td>Very short</td>
<td>Short</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Documentation required</td>
<td>Medium</td>
<td>Very low</td>
<td>low</td>
</tr>
</tbody>
</table>

Table 3-3. Plan Driven Methodologies Comparison.[91]–[94]

The comparison of methodologies in Table 3-3 exposes several factors why RAD is most suitable for this project. RAD requires less customer participation, which is useful in this project. Additionally, simplicity, time frame and required documentation are decisive factors, which lead the developer to choose RAD.

3.3. Summary

In this chapter the rationale to choice RAD as the methodology used in this project is presented. First, the main factors to determine a methodology for this project are presented. Then an analysis, where Plan Driven and Agile methodologies are compared, concludes that the first approach is the most suitable for this project. Finally, different Plan Driven approaches are compared, which leads to the selection of RAD.
Chapter 4  REQUIREMENTS

This chapter presents a summary of the artifacts produced during the requirements planning stage. By definition, the requirements analysis requires the participations of users, but in the case of this project the possibility of performing interviews, workshops or other techniques where the user is involved is limited. This is why the process of determining user expectations is based on research and on the guide provided by the Project Supervisor. Several related products and documents like object relational mappers [66, p. 2],[95–97], mobile applications with data synchronization features ([19]–[21], [40], [45]), synchronization and replication standards and tools ([21], [24], [39], [57]) and the official documentation for Android developers ([9], [82]) were analysed to identify potential stakeholders and requirements.

4.1. Requirements Elicitation

4.1.1. Statement of Need (SON)

Statement of Needs is a high level description of the problem addressed by the system. It provides an abstract view of the required product. SON is used as a starting point for requirements elicitation as it usually allows the developer team to identify an initial set of potential stakeholders and goals to be accomplished by the potential system [98].

The following statement of need is based on the information provided in the project statement:

The aim of this project is to implement a model-driven solution for generating efficient and correct-by-construction data-synchronisation layers for Android applications. In particular, the objectives of the project are to:

- Design a Domain Specific Language that enables engineers to model the data managed by a mobile application;
- Design and implement a model-to-text transformation that generates a server-side API through which client apps can interact with the server-side data storage;
- Design and implement a model-to-text transformation that generates a client-side API through which Android applications can manage their local data;
- Design and implement a layer through which the local data can be efficiently synchronised with remote data by orchestrating the server-side and client-side APIs.

4.1.2. Stakeholder identification

Before defining requirements, stakeholders should be identified. Stakeholders are individuals or groups of people who have direct or indirect interest in the system [98]. Stakeholder’s interest can arise from using the system or benefiting from it. As this is not a system that will be used directly by application users, the main stakeholders are developers. Nevertheless, other people are also affected by the use of it. Following these concepts, direct and indirect stakeholder have been identified.

A. Direct Stakeholders

In the context of this project, a direct stakeholder is defined as the person or group of people who interact directly with the system or the person who uses the system. The identified stakeholders are:

- **Application Programmer**: Android application programmers who will interact directly with the API provided by the system in order to create Android applications constitute the main stakeholder. In the case of the requirement process, programmer groups developing team members who program or test the application.
B. Indirect Stakeholders

- **Application Designer:** All developing team members who are involved in the design of the application. Examples of these are architects, modellers, etc.
- **Development Team Manager:** Development Team Manager is the person or group of people who manages the application development project.
- **End user:** The final user of the application.

4.1.3. Stakeholder requirements

This section summarizes functional and non-functional stakeholder requirements. These requirements have been defined from the statement of need, interviews with the Project Supervisor who assumed the role of the main stakeholder, and from the research performed from the sources specified in the introduction of this chapter.

To summarize information gathered during the requirements engineering process, a simplified format has been used, which contains the following information:

- **Identifier:** The two first sets of characters describe the type of requirement: SH stands for stakeholder and SY for system requirements and the second set of characters defines functional requirements as FR and non-functional requirements as NFR. The final set of digits is a unique numeral ID.
- **Description:** Short description of the requirement.

System requirements also contain the following information:

- **Acceptance criteria:** Acceptance criteria specify indicators or measures which will be employed to assess if the requirement is satisfied.

A. Functional requirements

Here is a description of the set of functions that direct stakeholders will be able to perform over the system in order to create an application with synchronization data capabilities.

**SH-FR-01**

**Description:** Programmers and application designers shall be able to model data managed by the mobile application using the DSL defined by the system.

**SH-FR-02**

**Description:** Programmers shall be able to generate Java classes from the data model in order to manipulate objects of the data schema.

**SH-FR-03**

**Description:** Programmers shall be able to generate Java classes to interact with and to configure the application database.

**SH-FR-04**

**Description:** Programmers shall be able to generate synchronization configuration artefacts required by central repository. This function shall be created as a SyncGateway configuration file where the synchronization logic should be defined.

**SH-FR-05**

**Description:** Programmers shall be able to perform creation, update and deletion operations over objects in the application database using API functions.

**SH-FR-06**

**Description:** Programmers shall be able to retrieve objects from the application database using filtering functions if needed.
SH-FR-07
Description: Programmers shall be able interact with the central repository from the Android application in order to authenticate users.

SH-FR-08
Description: Programmers shall be able to synchronize local application data with data in the central repository.

SH-FR-09
Description: Programmers shall be able to change the customize API configuration. Configuration values like network addresses of the central repository server and content authority of the application shall be contained in configuration files or classes¹ to make it easy to change them from their default values in order to facilitate testing and deployment.

SH-FR-10
Description: The system shall abstract out object metadata needed for synchronization operations from the programmer.

B. Non-Functional Requirements
The following non-functional requirements describe constrains on the services offered by the system.

SH-NFR-01
Description: The system shall provide an easy to learn and memorize API for programmers.

SH-NFR-02
Description: The system shall provide an API, which leads to readable code.

SH-NFR-03
The system shall provide an API, which prevents its misuse.

SH-NFR-04
Description: The time perform operations over the database shall not severely impact application performance.

SH-NFR-05
Description: The system shall provide the developer team manager facilities to successfully deliver applications on time and with little effort.

SH-NFR-06
Description: The system shall be able to be embedded in the application in order to facilitate application distribution.

SH-NFR-07
Description: The system shall be able to be installed in Eclipse Android IDE.

4.1.4. System Requirements
In this step of the process stakeholder requirements are further analysed to produce system requirements in order to define the specific functions that the final product will offer. The system requirements should provide a guide for the design and development process avoiding design and implementation assumptions in order to not interfere with these processes.

A. Functional requirements
SY-FR-01
Description: The system shall provide programmers a DSL to model the data schema that will be used to persist data in the application using the system.
Acceptance Criteria: The system shall offer developer and editor for modelling data schema where the DSL is used.

¹ Content authority values in an application identifies the content provider. This should be a fully-qualified class name (reduced to lowercase) to ensure uniqueness [99].
SY-FR-02
Description: The system shall validate data schema model before code generation.
Acceptance Criteria: The system shall not generate code from wrongly defined models.

SY-FR-03
Description: The system shall alert the programmer when errors are detected in the data model providing error messages which help programmer to find and correct errors.
Acceptance Criteria: The system shall provide error messages when the model is not correctly defined and shall prevent code generation while errors are not corrected.

SY-FR-04
Description: The system shall generate Java data classes and database access classes in order to provide the programmer with facilities to perform operations over data.
Acceptance Criteria: The system shall provide classes which should allow to perform all CRUD operation over data items.

SY-FR-05
Description: The system API shall offer the programmer the ability to create new data items and persist them in the application local database.
Acceptance Criteria: After calling to the persist function for new data item, their data should be able to be retrieved from the database using the system API or calling directly to database functions.

SY-FR-06
Description: The system’s API shall offer the programmer the ability to update data items contained in the database and persist changes in the application local database.
Acceptance Criteria: After calling to persist function for updated data item, data retrieved from the database should contain all changes.

SY-FR-07
Description: The system API shall offer the programmer the ability to delete data items from the application local database.
Acceptance Criteria: After call to persist function for deleted data items, their data shall not be retrieved from the database using querying functions.

SY-FR-08
Description: The system API shall offer the programmer the ability to filter data items from the local application database.
Acceptance Criteria: System API querying function shall retrieve only data items that comply with the provided filtering criteria.

SY-FR-09
Description: The system shall abstract the creation and update of the synchronization metadata in CRUD functions.
Acceptance Criteria: The system CRUD functions shall not require any kind of synchronization metadata in their parameters.

SY-FR-10
Description: The system shall provide the programmer with the ability to access the database directly.
Acceptance Criteria: System API shall provide a function to get the current database instance from classes created by the programmer such as activity classes where the application is defined.

SY-FR-11
Description: The system shall parse data from database to Java objects.
Acceptance Criteria: All database information should be available through the system’s API.
SY-FR-12
Description: The programmer shall be able to configure the name of the application local database and server connection URLs as well as configuration parameters of the syncAdapter and of the authentication manager.
Acceptance Criteria: System shall provide a configuration file where all configuration parameters can be changed.

SY-FR-13
Description: The system API shall provide the programmer with facilities to create, validate an authorize user accounts from the application in order to access to synchronization service.
Acceptance Criteria: The system API should provide functions to create and login user accounts for synchronization service.

SY-FR-14
Description: The system API shall provide functions to trigger and schedule synchronization process in order to incrementally synchronize information created and manipulated in the application local database with the central database.
Acceptance Criteria: System shall provide functions to synchronize data between local and remote databases and to schedule synchronization process for automated execution.

SY-FR-15
Description: Private information created with a user account shall be synchronized only to devices which are authenticated with that particular user account.
Acceptance Criteria: User accounts should not be able to retrieve private information from other accounts.

SY-FR-16
Description: Public information should be synchronized to all devices.
Acceptance Criteria: All user accounts should be able to retrieve public information from the central repository.

SY-FR-17
Description: The system API shall detect synchronization conflicts and resolved them if it is necessary.
Acceptance Criteria: The system shall abstract out conflict resolution process from the programmer.

B. Non-Functional requirements

SY-NFR-01
Description: System shall provide an easy to learn and memorize API.
Acceptance Criteria: Data management using the system API should be easier than using database functions directly.

SY-NFR-02
Description: System shall be able to be installed in Eclipse Android IDE.
Acceptance Criteria: System shall be delivered as an installable eclipse plugin.

SY-NFR-03
Description: The system shall provide an API which leads to readable code.
Acceptance Criteria: Data management using the system API shall be achieved in a more efficient way in terms of the number of instructions needed to perform a database operation compared to do the same operation using directly the database API.

SY-NFR-04
Description: The system shall provide an API that should prevent its misuse.
Acceptance Criteria: Data management should be easier and more intuitive using the system API than using database API directly.
SY-NFR-05

**Description:** The time to perform operations over the database should not severely impact application performance.

**Acceptance Criteria:** The evaluation application performance created using the system should not be significantly different from evaluation hand coded application performance.

SY-NFR-06

**Description:** The system shall provide the developer team manager with facilities to successfully deliver applications on time and with less effort.

**Acceptance Criteria:** The numbers of lines of coded necessary for creating an application using the system should be fewer than lines of code needed to create a hand coded application.

SY-NFR-07

**Description:** The system shall be able to be embedded in the application together with the database in order to distribute the application easily.

**Acceptance Criteria:** It shall not be necessary to install extra applications in devices in order to install applications created using the system.

### 4.1.5. Requirements Traceability Matrix

In order to ensure that all stakeholder requirements are accomplished by system requirements and to maintain the evolution relationship between them, the following requirements traceability matrix was created in the requirement analysis process.

<table>
<thead>
<tr>
<th>Stakeholder requirements</th>
<th>System Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH-FR-01</td>
<td>SY-FR-01, SY-FR-02, SY-FR-03</td>
</tr>
<tr>
<td>SH-FR-02</td>
<td>SY-FR-04</td>
</tr>
<tr>
<td>SH-FR-03</td>
<td>SY-FR-10</td>
</tr>
<tr>
<td>SH-FR-04</td>
<td>SY-FR-15</td>
</tr>
<tr>
<td>SH-FR-05</td>
<td>SY-FR-05, SY-FR-06, SY-FR-07, SY-FR-08, SY-FR-09</td>
</tr>
<tr>
<td>SH-FR-06</td>
<td>SY-FR-08</td>
</tr>
<tr>
<td>SH-FR-07</td>
<td>SY-FR-13, SY-FR-15, SY-FR-16</td>
</tr>
<tr>
<td>SH-FR-08</td>
<td>SY-FR-14, SY-FR-17</td>
</tr>
<tr>
<td>SH-FR-09</td>
<td>SY-FR-12</td>
</tr>
<tr>
<td>SH-FR-10</td>
<td>SY-FR-15,</td>
</tr>
<tr>
<td>SH-NFR-01</td>
<td>SY-NFR-01, SY-NFR-03, SY-NFR-04</td>
</tr>
<tr>
<td>SH-NFR-02</td>
<td>SY-NFR-01, SY-NFR-03, SY-NFR-04</td>
</tr>
<tr>
<td>SH-NFR-03</td>
<td>SY-NFR-01, SY-NFR-03,</td>
</tr>
<tr>
<td>SH-NFR-04</td>
<td>SY-NFR-05</td>
</tr>
<tr>
<td>SH-NFR-05</td>
<td>SY-NFR-01, SY-NFR-04, SY-NFR-04, SY-NFR-06</td>
</tr>
<tr>
<td>SH-NFR-06</td>
<td>SY-NFR-05, SY-NFR-07</td>
</tr>
<tr>
<td>SH-NFR-07</td>
<td>SY-NFR-02</td>
</tr>
<tr>
<td>Best Practice of API Development [100], [101]</td>
<td>SY-NFR-01, SY-NFR-03, SY-NFR-04</td>
</tr>
</tbody>
</table>

*Table 4-1. Requirements Traceability Matrix.*

### 4.2. Requirement Analysis

In this section a high level use cased diagram is presented and a description of this diagram is provided. The modelling of requirements allowed the developer to define interaction between actors and the system. It should be noticed that use cases in different levels were modelled but only one use case is
presented due to limited space in this document. In this use case the key capabilities of the system are summarized.

4.2.1. Use case model.

Figure 4-1 illustrates a high level use case diagram of the system. Actor is the developer who uses the systems to generate codes in order to build an application, the Android application that uses the system to interact with the local and remote database, and the remote database used to persist data remotely. The diagram presents a simplified view of the system with the main function provided by it.

![Figure 4-1. Use Case Diagram](image)
4.2.2. Use Case Description

Figure 4-1 illustrates 4 main capabilities of the system: generation of application code, management of user authentication, CRUD operations over database using the system API and synchronization data between application local database and server database. In this section only the latter use case description is presented due to space limitations.

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC-0-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Synchronize data between application local database and server database</td>
</tr>
</tbody>
</table>
| Actors:      | • Android application
• Database server |
| Trigger:     | Synchronization function is triggered by one of the synchronization triggers configured by the user. |
| Precondition:| 1. Application created using the system should be installed in the device
2. User account should be created.
3. User should log in before synchronization |
| Post condition| Application local database and remote database should contain the same version of data items |
| Main Success Scenario: | 1. The application validates user credentials with server
2. The application pushes new data items and updated items to the server
3. The server assigns access rights to data items
4. If conflicts exist, the server chooses winner version of the data item and marks conflict on it.
5. The server persists data items in server database
6. The server push new data items and updated items to application
7. The application pulls new data items and updated items from the server
8. If conflicts exist, the application chooses winner version of the data item and marks the conflict on it.
9. The server persists data items in local database |
| Exceptions   | 1. Credential not valid
2. Network connection is not available or fails during process
3. Server does no respond |

4.3. Summary

This chapter presented a summary of the artifacts produced during the requirements planning stage. First, section 4.1 introduces the phase of requirements elicitation where the statement of need and the stakeholder identification are specified. Then, stakeholder requirements are extracted through the analysis of the statement of need and similar projects. In the last part of this phase, stakeholder requirements are refined and elicited in order to produce functional and non-functional requirements of the system. In the second phase, the requirement analysis, system requirements are modelling using use cases.
This chapter analyses the design approach used in this project. First, the design process followed in this project is described and then the process itself is documented.

5.1. Design process

When requirements do not change too much, defining the architecture at an early stage provides for considerable predictability making it easier to implement the system and reduce risk [102, Ch. 15]. Nevertheless, in the case of this project, where the developer knowledge about the used technologies is limited and experimentation is an important factor to understand these technologies, this approach could be inflexible and could limit invention and creativity. In contrast, the lack of up-front planning and commitment can result in chaos [102, Ch. 15]. These factors lead the developer to take a mixed approach, proposing an initial architecture before starting the implementation process, but maintaining the possibility of refactoring the architecture when required.

During the development of this project, an initial architecture was proposed emphasizing on the definition of the central patterns to be used. This early architecture was the guide for the developing process providing early problem understanding and analysis. Nevertheless, this initial architecture was refactored when new functional or quality requirements emerged or mistakes where found. As Booch claims: "The life of a software architect is a long and rapid succession of suboptimal design decisions taken partly in the dark" [103], and during the process of implementation the developer often can realize that his or her initial decisions are not the most optimal or efficient.

Considering these factors, Figure 5-1 illustrates the process that was followed during the development of the system. On the basis of an initial architectural design, a prototype with the most important components was implemented. This initial prototype provided the developer with the ability to evaluate the initial design and recognize better approaches to implement the system. In the next iterations, not only new components were added to the prototype, but also already implemented components were improved and reevaluated and the system architecture was refactored when necessary.

![Figure 5-1. Design Process.](image-url)
5.2. Architecture

The main goal of defining a software architecture early in the developing process is to achieve a structured solution that meets functional and non-functional requirements, and optimize system quality attributes [102, Ch. 2]. In order to achieve this goal, the Attribute-Driven Design Method approach [102, Ch. 17] guided this process. First, the most important quality attributes for the system were identified. Then, the tactics that can be used to achieve these quality attributes were analysed and prioritized. Finally, these tactics were used to create a design for the system. This design was tested and refactored several times because possible improvements were identified at different stages of the process. In this section the final design is presented.

5.2.1. Quality Attributes and Tactics

In this section, the most important quality attributes in the design of the system are analysed. Non-functional requirements described in section 4.1.4.B were analysed in order to select quality attributes that implement, extent and complement these requirements. Finally, these attributes were prioritized in order to provide a guide when trade-offs where necessary.

Each quality attribute is presented together with its rank and a set of tactics used to achieve them. Tactics are grouped by the main quality attribute that it accomplishes, although it can support others. The most important quality attributes which drove the design are:

A. Usability

As discussed in system non-functional requirements, two of the main goals of the system are to provide developers with an easy to use and hard to misuse API. This quality attribute was used to achieve the following characteristics in this system.

- The system should be easy to learn.
- The system should minimize errors caused by misuse of the system and their impact.
- The system should be able to be adapted to user needs.
- The system should provide user confidence on what the system does is correct.

**Rank:** High

i. Usability Tactics

Tactics used to achieve usability are:

Maintain User Model.

The system is designed to maintain the user model of the Android Software Development Kit (SDK). Android developers usually use natively supported Android features but the system introduces technologies that are not widely used by Android developers, for example NoSQL databases. The system is designed to provide an API which maintains the Android SDK task model when it is possible by abstracting other task models.

Maintain System and Task Model

The system is designed to maintain a system model across all its components in order to make it easier for the user to determine the system behaviour using appropriate feedback. The use of error messages and Java doc are examples of this.

B. Modifiability

Modifiability focuses on the ease with which a system can be modified to correct faults or to be improved [104]. As discussed in section 2.3.2, NoSQL databases in the context of mobile application is a relatively novel approach which is constantly under evolution. It makes it important for the system to be flexible enough to take advantage of changes in an easy way.

**Rank:** High
i. **Modifiability Tactics**

Tactics used to achieve modifiability are:

**Reduction of Modules Size**

The system was refined into small components in order to make it easier to introduce changes. Making changes in small components is easier and faster than in big ones.

**Increase of Semantic Cohesion**

When services provided by components did not serve the same purpose, they were placed in different components or a new component was created. This also makes it easier to introduce changes in one service by avoiding side effects in other services.

**Coupling Reduction**

In order to reduce coupling between components, explicit interfaces were used in order to avoid the propagation of changes across modules. These interfaces were designed to abstract the details of implementation of the service that each component provides.

C. **Functionality**

Even though functionality does not determine architecture [102, Ch. 4], it is a decisive factor to take into account when the design is done. Functional requirements were achieved by assigning responsibilities to architectural elements considering the architecture support to other quality attributes.

**Rank:** High

i. **Functionality Tactics**

Functionality and quality attributes and their tactics are orthogonal but closely related [102, Ch. 4]. The mapping of functional requirements of the system onto architecture elements determined the support of the architecture for quality attributes.

D. **Performance**

Applications created with the provided solution demonstrate a significant performance handicap compared to manually developed applications with similar functionality. Besides the system introducing an additional layer in the application, its impact in performance should be minimized.

**Rank:** Medium

i. **Performance Tactics**

Tactics used to achieve performance are:

**Control Resource Demand**

In order to increase performance, the demand for resources is limited. For example, the processes that mostly affect performance are CRUD operations to database. In order to reduce the number of operations over the database a data cache can be introduced. This data cache can allow developers to interact with data deferring CRUD operations over database when it is possible. Another example is the use of the Android SyncAdapter framework in order to optimize the use of device resources like processor, memory, battery and network resources.

**Manage Resources**

As well as mobile device resources such as processors, memory or network connections are limited, their use should be carefully managed. In order to optimize the use of these resources different tactics were used. An example is the use of concurrent processes using Android services to perform long running operations such as server authentication and synchronization. Another example is the allocation of most resources demanding operations in the server side of the system, where resources can be increased more easily. The schedule of the synchronization process using the Android syncAdapter framework capabilities is another example of how the use of network resources are managed in order to optimize its use.

E. **Security**

The system should be able to protect data, providing users the ability to access their data, while the system protects this data from other users. User credentials should also be managed without compromising their confidentiality.
i. Security Tactics

Tactics used to achieve performance are:

Authenticate Actors

The system provides functionalities for user authentication using passwords and authentication tokens. This allows developers to authorize the user to access and modify only the data assigned to that specific user. Actor’s authorization also implements other security tactics mentioned by Bass and Clements which are closely related: actor’s identification, authorization and access limitation [102, Ch. 9].

5.2.2. High Level Architecture

As the aim of this project is to provide a data synchronization layer for Android applications, it is useful to analyse how the system will be integrated in a common data centred Android application before discussing the architecture of this system in detail. Figure 5-2 presents a general architecture used for Android applications [105] and which layers of this architecture will be contained in the system. The goal of the system is to provide an API that abstracts the data layer of this architecture. Additionally, in order to provide facilities to the developer, the API also abstracts the background logic layer to provide authentication and synchronization management services.

Figure 5-2. Android Application Architecture

Figure 5-3 shows a static view of the system architecture and how the system interacts with the logic layer in an Android application. The system consist of three main components: an API code generator, the Android application API (SyncAPI), and the server application where the central data repository resides. The services provided by each of these components and their sub components are described in this section.
Figure 5-3. High-Level Architecture
5.2.3. Component Design

In this section each of the components introduced in the high level architecture are detailed.

A. API Generator

The API Generator component gives the developer the ability to generate the DataAPI component and a set of different configuration files for the Android Application and the server side of the system from a specified data model. This component was split in 3 sub components: the model editor, the model validator and the code generator. Figure 5-4 illustrates an activity diagram which shows the workflow of this component. The developer creates a data model for his or her application using the domain specific language defined by this component using the Emfatic editor. Then this model is validated. If the model contains errors, the system provides error messages in order to make easier the correction of these errors. Only when the model is correct, the system creates all files needed for the Android application to run the SyncAndroid system and the synchronization configuration file for SyncGateway.

Figure 5-4. API Generator Activity Diagram

i. Model Editor and Domain Specific Language

The Emfatic editor is a feature-rich text editor, which can be used by the developer to create a data model in a textual form using the Emfatic language. As it was explained in section 2.5, Emfatic language is used to model the data structure of the Android application. The API Generator component provides developer the ability to generate the DataAPI and a set of different configuration files for the Android Application and the server side of the system from a specified data model. The use of Emfatic language allowed to reduce implementation overhead, proving a way to represent a data model in a textual form using a simple language in a feature-rich text editor.

After considering the implementation of a customized meta model for the system, the use of the Ecore meta-model was the most suitable choice for the project. The use of Ecore provides several advantages to this project, some of the most important are: Ecore is widely used and known by developers, it is a mature and extensively tested meta model and it contains all the necessary concepts needed for this project (details are discussed in section 2.5). Figure 5-5 illustrates a simplified subset of the Ecore meta-mode. This diagram shows the main elements of Ecore needed to represent a data model for the SyncAndroid System: classes, attributes, data types and reference between classes.
In order to provide system’s users the ability to define concepts of the data synchronization domain, a list of possible annotation was defined for the DSL. Listing 6-2 shows a simple example of a data model represented by the DSL. It is important to note that the DSL is based in Pongo’s (Java POJO generator for MongoDB [75]) DSL as explained in section 2.5.

```
package com.york.cs.todolite2.document2;

@db
class TodoDB {
    val ListTasks[*] listTasks;
    val Task[*] tasks;
}

@document(sharing="shareable")
class ListTasks {
    attr String title;
    ref Task[*] tasks;
}

@document(sharing="shareable")
class Task{
    attr String title;
    attr boolean checked;
    @image
    attr String image;
    val Location[1] location;
}

class Location{
    attr String name;
    attr String address;
}
```

Listing 5-1. DSL ToDo Example

The main concepts of the DSL are explained below:

- **Package**: The package defines the Java package where all the generated Java classes will be placed. The name of the package is also used to define the content authority and the account type of the Android application.
- **Database**: The database concept represents the main container of data. Only one can be defined in a data model using the @db annotation. The Database must refer to database documents using containing references only and no attributes could be defined for this class.
- **Document**: Documents class represents the concept of document in NoSQL databases. It is an important concept because it defines the minimal unit of data to be synchronized. This means that the control version keeps versions of each document, and if an attribute or a
class contained in the documented is updated, a new version of it is created. All classes contained in the database class are considered documents. Also, annotation @document can be used to define it.

Sharing attribute defines if a document can be shared between accounts. Possible values are: private if these data type are only accessible by the account which was used to create it. If “sharable” is used, the user will be able to share the documents created using this class with other accounts. Finally, if value “public” is used, this document will be available for all user accounts.

- **Image and attachment**: Attachments are managed separately from other attributes in order to optimize synchronization. Image annotation creates a Bitmap attribute that can be used to attach images to a document using Bitmap objects. Attachment annotation creates an InputStream attribute that can be used to attach other kind of files, but in this case the developer should care about parsing the file. It is important to notice that attachment and image attributes can be only defined in document classes and not in classes contained in them.

- **Val reference**: Val reference defines a content relationship between classes. Document classes can contain any kind of classes which are not documents. No document classes also can contain classes in it. It is important to notice that val references cannot be defined between document classes.

- **Ref reference**: Reference is used to link documents. In the example of Listing 6-2 a reference is defined between the ListTask classes and the Task class, meaning that a ListTask can contain any number of classes and a Task can be contained by only one ListTask. An important limitation of this concept is that references can be defined only from document classes or from a no document class (any class contained in a document) to a document class. References between no document classes and from a document class to a no document class are not allowed.

Emfatic language also supports concepts like inheritance and enumeration types. More details about these and other concepts can be found in [106]. An important limitation of the DSL is that custom data types (i.e. data types other than the built-in String, Integer ...) are not supported.

Finally, when the model has been defined and saved as an emf (Emfatic) file, SyncDocGenerator class creates an Ecore model from it and calls the modelValidation component. SyncDocGenerator class manages the execution of all subcomponents. Figure 5-6 illustrates the structure of the API Generator component.

**ii. Model Validator**

Before code generation, it is necessary to validate the correctness of the data model. The Epsilon Validation Language (EVL) is used to specify and evaluate a set of constrains defined specifically for the domain of the system. These constraints allow to produce a model that fulfils all the specific requirements of NoSQL data model and Couchbase database that are not defined in the Ecore meta model. Some examples of these constrains are listed below:

- Attribute names cannot start with the character “_” (underscore.). This constraint is defined by Couchbase Lite where only metadata of documents can use this kind of name.
- References across documents should have as target a document class and as source a document class or a class contained in a document.
- Attachment and Image annotations can only be defined in document classes. Classes contained in document classes cannot contain this kind of attributes.
- References between document classes and no document classes should be of type containment. Containment references must by defined using the “val” keyword.
- “@db” and @document annotations should be defined only for classes, never for attributes.
To validate these constraints SynDocGenerator class uses the modelValidation EVL file where constrains are specified. Only if no errors are detected, SynDocGenerator class continues with the code generation. If errors are detected, SynDocGenerator displays error messages.

iii. Code Generator

This component generates the DataAPI for the SyncAPI component and configuration files for the Android application and for the server. To achieve this, SynDocGenerator uses two EGX\(^1\) programs: coordinator.egx and coordinatorConfigFiles.egx.

The coordinator.egx program generates Java classes that will constitute the DataAPI subcomponent of the SyncAPI component using a set of EGL\(^2\) templates as it can be seen in Figure 5-6. These templates generate different types of classes which provide different services to the developer:

- `eclass2appclass.egl` template creates a class called Application which provides services to initiate the database and to configure the API.
- `eclass2dbclass.egl` template creates a class which represents the database, and provides services to access database collections and to persist data to the database engine.
- `eclass2collectionclass.egl` template creates a collection class for each document class defined in the model. This class provides services to aggregate and retrieve data from the database.
- `Eclass2class.egl` templates generates java classes for document and no document classes specified in the model representing the structure of the data that will be contained in the database. These classes provides getter and setter methods for the attributes and references defined in the model.
- `Eenum2enumeration.egl` template generates Java enumeration classes for enumeration classes specified in the model.

---

1. EGX programs are coded using EGX language which is a coordination language for EGL templates.
2. EGL is a template-based code generator which support the generation of text from models.
The coordinator.egx program generates configuration files and configuration snippets for the Android application and the server using a set of EGL templates as it can be seen in Figure 5-6. These templates generate different types of codes which allow the developer to configure the system:

- **Android application configuration files:**
  - `eclass2manifest.egl` template creates a code snippet which contains all the permissions and declarations of syncAdapter and authentication services, content provider and activities that the developer should declare in the Manifest.xml file of the application in order to use the API.
  - `eclass2authenticator.egl` template creates an xml where the configuration of the user manager API component is located.
  - `eclass2sync_adapter.egl` template creates an xml where the configuration of the synchronization manager API component is located.

- **Server configuration files**
  - `serverConf.egl` template creates a configuration file for the SyncGateway server. This configuration file contains the IP address of the servers and the database configuration of the database that will store the data in the Couchbase server. It also contains the sync function which is a JavaScript function used by the Sync Gateway to coordinate the synchronization of data. Every time a new document, revision or deletion is added to a database, the logic defined in the sync function is used to perform different operations in the metadata of these documents [107].

**B. Synchronization API (SyncAPI)**

The synchronization API is the main component of the system. It allows the developer to perform CRUD and synchronization operations over the database without worrying about all the complexities of the management of the metadata needed to control versioning of data items and the management of Android accounts and synchronization services. In order to deliver these facilities to the developer, this component has 4 subcomponents: the DataAPI, the object-relation mapper, the synchronization manager and the user manager API. These components are designed to be used in an Android application as a data layer. The details of each of these components are discussed below:

i. **DataAPI**

The DataAPI component provides a simple object oriented API, which can be used by Android developers to store, update, delete and query for Java object. The main advantage that this API provides is that it abstracts the management of synchronization metadata and the complexities of working with database documents. As the API is generated from the data model specified by the developer as explained in section 5.2.3.A, its structure will vary depending on the data model. Figure 5-7 illustrates a simple example of an API generated using the system¹. All the classes generated for these components extend the functionalities of different classes of the Object Relation Mapper component.

One application class will be generated in all cases. As this class extends Android Application class, Android creates an instance of it when the application is started and it is kept in memory as long as the application process is running [108]. This allows keeping a single instance of the database and of the cache during the life cycle of the application. This class provides methods to initiate the database when the application is started, to get the database object from other classes and to configure the API.

---

¹ The model used to generate this API can be seen in Listing 5-1. Dsl ToDo Example on page 45. A more detailed class diagram with operation and attributes can be seen in Appendix B-1
Also, only DB class will be generated. It represents the database concept and as explained before only one instance of it will exist during all the lifecycle of the application. This class provides methods to persist data stored in cache in the database and to get the collections of documents that the database contains. In Figure 5-7 this class is represented by class TodoDB. This class extends CouchDb class which is part of the Object Relation Mapper component.

For each document class defined in the data model, a collection class is generated. These classes provide services to add and remove document objects to the database and to retrieve data from it. It is important to notice that when an object is added to a collection it is not persisted immediately to the database, it is added to the cache instead. Also when objects are retrieved using these classes, it first queries the cache. If there is not an instance of the object in cache, the document that contains the object is recovered from the database and it is parsed to a Java object and added to cache.

Collection classes also provide services to retrieve collections of objects which can be filtered using objects’ attributes. These facilities allow developers to interact with collections of objects using similar methods as a regular Java List or Java Iterator does. These classes extend the DbCollection class which is part of the Object Relation Mapper component. Examples of these classes are ListTaskCollection and ListTasksCollection in Figure 5-7.

Figure 5-7. TODO DataAPI Example

For each document class defined in the data model a Java class is created. These classes provide setter and getter methods for all attributes, objects and relations contained in the classes. These classes also contain metadata of the documents. These classes extend DBObject class which is part of the Object Relation Mapper component. Examples of these classes are Task and ListTasks in Figure 5-7.

Finally, for each class that is contained in each document class, a class is generated. These classes provide setter and getter methods for all attributes and objects contained in them. Examples of these classes are Location class in Figure 5-7.

ii. Object-Relation Mapper

This component provides different services to support the DataAPI component. The design of this component is based on the project Pongo [75], which provides an API for the NoSQL MongoDB database. This approach was adapted to be used with Couchbase Lite and to manage synchronization metadata.
Classes of the DataAPI component inherit the functionality of three core classes: CouchDB, DBCollection and DBObject. These classes provide services to perform create, read, update and delete operations over the database. Additionally, this component implements a data cache in order to reduce the necessity of retrieving data from the database multiple times if an object already has been read from the database as explained before.

Figure 5-8 is a simplified class diagram which illustrates this component. CouchDB class provides services to initiate the database, write data to database and retrieve documents contained from the database. The write operation is performed by the sync method, which synchronizes the data contained in the cache with the database. This class also references to database collections, which group database objects of the same type. A collection is often described as the equivalent of a table in a relational database [109]. As Couchbase Lite database does not support the collection concept as other NoSQL databases do (e.g. MongoDB), it is implemented by the DBCollection class.

DBObject class abstracts the notion of documents in NoSQL databases. A root DBObject\(^1\) contains a Java map structure which constitutes a document in Couchbase Lite. This map can contain primitive data types or complex data structures in it. Primitive data types are parsed from and to map directly by this class but when collections of primitive data types are stored in the map, PrimitiveList class is used. This class allows the developer to access to collections of primitive types abstracting the operations of parsing types from and to objects.

As documents can contain complex data types in it, DBObject also can contain single DBObjects or collections of them. When a single DBObject constitutes a variable in its parent DBObject, it is parsed from and to a nested map contained in the parent document. When a variable contains a collection of DBObjects,

\(^1\) Each document persisted in the database is represented with a DBObject, called root DBObject in this chapter. A root DBObject can contain primitive variables or other DBObjects.
DBList and DBListIterator are used to perform parsing operations while the developer iterates through the collection. It also provides services for adding and removing DBObjects from their containers.

DBObject also abstracts from the developer the management of the metadata needed to implement the synchronization process. A Universal Unique Identifier (UUID) [110] is created with each DBOBJECT, which is used to uniquely identify documents in both, the device and the server databases. When a synchronization user account has been created, each document also contains the ID of the user in order to assign documents to a particular user account making it inaccessible to other users. Additionally, metadata for managing revision or version IDs and the deletion state of the document are managed by this class. Couchbase also defines a special property called “_attachment” which contains metadata about document attachments in order to manage them separately from structured data. DBOBJECT provides services to store data associated with the document as attachments in order to optimize the synchronization process [111].

DBCollection class is the link between CouchDB and DBOBJECT class. It provides services for retrieving, deleting, adding and updating documents from or to the database. Additionally, when a document is retrieved and parsed to a root DB Object from the database or a new root DBOBJECT is added to the collection, DBCollection also adds that root DBOBJECT to the data cache using the ClassDBFactory class. In order to provide querying functionalities to the developers, DBCollection class uses the generic class DBList. This class allows the developer to interact with collection of DBOBJECTs through an iterator provided by the DBCollection class.

Couchbase Lite does not provide facilities to create references between documents. DBRef class provides this service allowing DBOBJECTs to reference other root DBOBJECTs. This class provides services to create and resolve references.

DBCollection class uses QueryProducer and MapExplorer classes in order to provide developer facilities to retrieve filtered collections of root DBOBJECTs. Couchbase Lite provides only basic functionalities to filter documents using a map reduce model [112]. These functionalities have been abstracted using the QueryProducer class. For each variable contained in a DBOBJECT, a QueryProducer is provided. QueryProducer objects contain an address defined in dot notation [109] which allows to create database indexes for each variable defined in DBOBJECTs. This notation is implemented by the MapExplorer class, which allows to filter documents using their attribute values that can be complex objects or primitive types. The filtered collection of documents are delivered to the developer as a DBList object.

Finally, in order to provide facilities for using the data managed by the API, also an abstract class called EntityAdapter is provided. This class extends Android Adapter class to provide a bridge between a ListAdapterView1 and the underlying data for that view2. The Adapter provides access to the data items. The Adapter is also responsible for making a View for each item in the data set and provides facilities to automatically update the view if changes are made in the database.

### iii. User Manager

Several authors and developers agree that account management in Android is a complex and poorly documented topic [9, p. 144], [113, Ch. 17], [114], [115]. Nevertheless, account management tools provide several facilities for the synchronization process. It is even a compulsory requirement for implementing the Android syncAdapter framework, and in this project it is essential for the authentication and synchronization

---

1 Listview is an Android view (see footnote 2) that shows items in a vertically scrolling list. The items come from the ListAdapter associated with this view.

2 A view is a building block for user interface components. Views occupy a defined area on the screen and is responsible for drawing and event handling. Additionally, views are the base class for widgets, which are used to create interactive UI components.
processes in the server side. An Android developer should perform several things in order to create a user account, a summarized list of these steps is presented below:

- To configure account management permissions in the Manifest configuration file.
- To define an account authentication service component in its Manifest configuration file. For this configuration a metadata declaration that refers to a resource describing the application’s account type must be created, using an xml document.
- To implement the authentication service that returns an instance of a subclass of Android’s account authentication template class.
- Finally, to implement user interfaces of registry and login of accounts.

The user manager module abstracts this process from the developer giving an API that provides services to manage accounts without worrying about complex interactions with Android services, the application server or authentication tokens. This module provides an API for performing basic operations over user accounts through the UserManager Class. The principal services provided by this class are:

- userSignUp (User): This method takes as parameter a User object which contains user’s name, email and password and creates a user account in the device and in the application server. This method also creates a Profile document in the database which can be used to share data with other users. Additionally, if data is created in the database before the account is created, it updates all documents contained in Couchbase Lite in order to update the metadata field ownerId, needed for the synchronization process. If there was an account before the creation of the new user account, all data in the database is purged, meaning that it is deleted from the local device database but not from the central repository. Due to limited space only this process is briefly summarized in Figure 5-9.

- userLogin (User): This method takes as parameter a User object which contains at least user’s email and password and validates the user credential in the server side. This method also creates a Profile document in the database for data sharing purposes. If credentials are
correct a user account is created in the device. It also updates the metadata field owner if it is needed, following the same conditions as the process described before.

- **removeCurrentAccount():** This method deletes the current users account from the device and purges all documents in the database. The system only supports one user account. If a user logs in with a different account from the current one, this method is called by the API.

- **Finally,** some utility services are provided for the developer in this API, `getCurrentDeviceAccount` method returns the account data of the current user. Method `existsAccount` returns as true if a user account has been created in the system or false if there is no account.

Figure 5-10. User Manager Class Diagram

Figure 5-10 illustrates a high level class diagram of the component. AuthenticatorApp class is an implementation of `AbstractAccountAuthenticator` class provided by Android as a template for implementing “AccountAuthenticators” [116]. The implementation of this class also requires to implement a service. AutheticatorService class provides the required service, and performs long running operation in the background in order to exchange data with the application server using REST web services.

Additionally, some model classes were defined for this component. The user allows the developer to create a user object, which is used as a parameter in some methods of the API and in the web service provided by the server application. SessionSync defines the object returned by the application after a SingUp or LogIn request. Finally, Profile is a DBOobject1 class used to persist user account data in the database.

To implement these services in an android Application the user should define user interfaces for registering or creating new user accounts and for login in existing user accounts. In order to facilitate the use of this server, user interfaces are provided for these purposes. These interfaces are defined in two Android activity classes which make use of the UserManager API as it can be seen in Figure 5-11. Additionally, the layout definition of the interfaces are provided with the library as XML files and code snippets are generated to configure them in the application.

---

1 DBOobject concept is detailed in section i.
iv. Synchronization Manager

Although a custom module for managing data synchronization can be implemented, using the sync adapter framework provided by Android offers several advantages. This framework provides features like [117], [118]:

- Plug-in architecture totally supported by Android.
- Automated execution that can be triggered based on a variety of criteria like data changes, periods of time or defined time of day. Additionally, if a synchronization process is unable to run, it is added to a queue managed by the operative system which will run it when possible.
- Automated network availability checking.
- Improved performance of battery. Since the scheduling of the synchronization process is delegated to the operative system, it will be scheduled together with other data transfer requests from other apps in order to use the battery and network resources in a more efficient way.
- User authentication is integrated with Android account management facilities.

An Android developer should perform several things in order to use these features, a short list of these steps is presented below:

- To create and configure an authenticator. In the case of this system, the module user manager provides the authenticator service.
- To create and configure a Content Provider.
- To implement and configure a syncAdapter class.

Figure 5-12 illustrates a class diagram of this component where the classes used to implement the sync adapter framework.
SyncManager class is the API provided to the developer to schedule and trigger the synchronization of the database. This class provides the following methods:

- syncWithNetMsg() method set up sync adapter to trigger the process when it receives a network tickle from the operative system. This option is used to run the system adapter automatically and usually it is used with interval-based sync scheduling [119].
- syncScheduledPeriodically() takes as parameter an interval of time in seconds in order to schedule the synchronization process to be triggered at regular intervals of time.
- syncCurrentUserNow() allows the developer to trigger a synchronization process on demand.
- isSyncActive() can be used to know if the synchronization process is active or not.

Android’s sync adapter framework requires the implementation of a content provider. Nevertheless, the data management API provided by the SyncAPI component does not use this feature because the content provider is designed to be used with SQLite and not with a NoSQL database. To meet this requirement, a stub content provider was added to the system using ContentProviderApp class. This class implements Android’s ContentProvider class but all its methods return null or 0. This is the standard procedure to implement sync adapter with alternative storage mechanisms [120].

SyncAdapter framework also requires the implementation of a bound service in order to run long running network and synchronization operations in the background. This service is implemented by class SyncAService and it allows to trigger the synchronization process, not only by the application, but also by the operative system if the synchronization is scheduled using its facilities.

The service mentioned before invokes onPerformSync() method in SyncAdapterCB class when it needs to trigger a synchronization process. This method encapsulates the data transfer code and can be called by the app or by the operative system. This class also interacts with the user management module in order to get the authorization token required by the server.

SyncAdapterCB uses Couchbase Lite facilities to perform the synchronization process. Couchbase facilities transfer data changes between the local database in the device and the remote database in the server using its own algorithm. In short, this algorithm starts 2 replication processes, one for pulling data to the server and one for pushing data from it, which will stay active until all changes have been replicated. A deep explanation of this algorithm and its source code can be found in [122].

Also it is important to mention how synchronization conflicts are managed during this process. Couchbase Lite uses the Multiversion Concurrency Control (MVCC) method to manage conflicts when a document is updated by multiple writers. This version control system is widely used, examples of this are Git, Subversion and WebDav [123].

In order to implement MVCC, every document created in the database has a metadata field called “_rev” which contains a revision ID. This ID is updated every time a document is saved. This metadata is used to create a revision history in a tree data structure allowing to keep different versions of a document when there is conflict in order to solve conflicts if self-defined conflict solution logic is to be implemented. Nevertheless, a “winner version” should be selected in order to respond to queries performed to the database. To be precise, Couchbase Lite uses the following rules to select a winner version between multiple conflicting versions [124]:

- The winner is the undeleted leaf version on the longest revision branch.
- If there are no undeleted leaf versions, the deletion on the longest branch wins.

---

1 A bound service allows components of application like activities to bind to the service, send requests, receive responses, and perform interprocess communication. [121]
Finally, the winner is the one whose revision ID sorts higher in a simple ASCII comparison when there's a tie.

C. Central Repository

The application server provides a central repository where Android applications can back up their data. To provide this service it uses the facilities provided by the Couchbase Lite framework. As it can be seen in Figure 5-3, this component has three subcomponents: the application server which provides authentication and authorization services, the SyncGateway server which controls the data synchronization process, and the Couchbase server which provides a NoSQL document database. These components will be discussed in this section.

i. Application server

This component provides services for user creation and authorization. The design of the web services is illustrated in Figure 5-13.

User web service allows Android application to register new users, login already registered users and get an authorization token needed to synchronize data using the SyncGateway server. Due to limited space, Figure 5-14 only illustrates the user login process in the web service. As it can be seen in Figure 5-3, the data exchanged between Android application and these modules is done using Restful web services. Figure 5-13 also illustrates the entities used to exchange data: a user entity which contains information needed for user creation and authorization, and the entity SessionSync which contains the session ID used as an authorization token by SyncGateway. Additionally, possible faults have been defined for the following cases:

- NotAuthorized is returned in case that user credentials cannot be validated
- IncorrectInformation fault is returned in case that the user data provided is not complete
- UserConflict is returned when an attempt to create a new user with an email already registered is detected and the provided password cannot be validated.
- InternalError is returned when any internal error is detected. Examples of these are cases where connection with SyncGateway or Couchbase server is lost.

Figure 5-13. Web Service Design
To provide these services, this component uses authentication services provided by SyncGateway. The communication between these two components is also done using RestFul services.

**Figure 5-14. User Login Activity Diagram**

ii. SyncGateway and Couchbase server

SyncGateway and Couchbase server were discussed in section 2.3.3 where their main functionalities and benefits were detailed. Nevertheless, it is necessary to discuss some concepts directly related to the design of the system.

Sync Gateway server enables Couchbase Server to act as a synchronization endpoint for this system. Sync Gateway runs an HTTP listener process that provides a passive synchronization endpoint and uses a Couchbase Server bucket as persistent storage for all database documents. The Application interacts with SyncGateway using the Sync Manager. This interaction is done using the Sync REST API provided by SyncGateway as it can be seen in Figure 5-3.

Before performing a synchronization process, a user account must be created in the SyncGateway server. This is done by the application server using SyncGateway Admin REST API to manage user accounts and sessions. By default, the Admin REST API is reachable only from the application server for safety reasons.

Finally, the data synchronized using SyncGateway is persisted in the Couchbase server. As it can be seen in Figure 5-3, this server is accessed only through SyncGateway.

5.3. Summary

This chapter summarizes the design process. Section 5.1 introduces the design process followed to achieve the final design of the system. Section 5.2 analyses quality attributes for the design of the system and the tactics used to achieve them. Then, the high level architecture of the system is introduced and each of its components is described.
Chapter 6
IMPLEMENTATION

The implementation of the SyncAndroid system is described in this chapter. The implementation followed an iterative process because of the selected development methodology. In each iteration new components were added, and existing components were improved. Nevertheless due to space limitations and to make a more comprehensive description of the components, only the last versions of the components are illustrated here. Before describing the implementation of the components, the preparation and configuration of the development platform will be illustrated.

6.1. Preparation

As it can be seen in section 5.2.2, where the high level architecture of the system is described, different tools were used to achieve the systems requirements. Therefore, prior to actual coding, it was necessary to run a preparation phase in order to prepare a developing environment for the system and simulate a developing environment that could be used by developers using this system to create Android applications.

6.1.1. IDE Preparation

As it was concluded in section 2.6, the most convenient IDE for developing the system is Eclipse. To develop the SyncAndroid system, a list of plugins were installed over Eclipse Kepler. These plugins are presented below:

- **Web Tools Platform (WTP):** This project extends Eclipse to provide facilities for web and enterprise (J2EE) application development. WTP contains a wide variety of tools, but the main tools used in this project where the Java JEE tools and the server tools which provide facilities for developing Java EE artifacts [81], [125]. These tools were essential for developing the application server component of the system. These tools can be installed in any version of Eclipse. Nevertheless, the installation of these tools can be complex because of the big amount of dependencies with other projects and subprojects that they have. This is why it is widely recommended to use the Eclipse IDE for Java EE Developer version which already have all the tools needed for developing the application server for this project. The version used was Eclipse IDE for Java EE Developer is SR2.

- **Epsilon:** The facilities that Epsilon provides to this project were already discussed in sections 2.5 and 2.6. These tools were installed using the update sites that Epsilon official download page provides. The interim 1.1 SR1 version was installed along with Emfatic.

- **Android Development Tools (ADT):** This plugin provides an environment to build Android applications. This was installed with the Android SDK. The version used was 23.0.2.

6.1.2. Couchbase Lite Tools.

As it can be seen in section 5.2.2, the system uses a Couchbase database server and a SyncGateway server to provide a central data repository. The Couchbase open source project provides community editions of both servers under an Apache 2.0 license. The versions used during the implementation project were


3 [https://dl-ssl.google.com/android/eclipse/](https://dl-ssl.google.com/android/eclipse/)
2.2.0 64 bits Windows for Couchbase database server and 1.0.1 64 bits Windows for SyncGateway. These servers are provided as executable installation programs which can be easily installed in Windows. Default configurations were used.

It is worth noticing that Couchbase also provides the service Couchbase Cloud which is an online environment that can be used to test Android applications that use Couchbase tools. This tool was not used in order to access to all API interfaces that SyncGateway and Couchbase server provides in order to perform testing operations.

6.2. Components implementation

This section describes the implementation of each of the components that comprises the SyncAndroid system.

6.2.1. API Generator implementation

As discussed in section 5.2.3.A, the API Generator component provides developers with the ability to generate the DataAPI component and a set of different configuration files for the Android Application and the server side of the system from a specified data model. To implement this functionality an Eclipse plugging was created. This plugin allows the developer go through the steps specified in Figure 5-4.

This component was implemented as an extensible Eclipse plugin. Therefore a list of plugin projects were configured as dependencies and some extensions were created to implement these functionalities.

The main types of dependencies that were defined for the plugin are:

- Epsilon plugins. These plugins provide functionalities for model validation and code generation using the EOL, EGX, EGL and EVL languages.
- Eclipse JDT and UI: These plugins provide the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins and Eclipse user interfaces.

Although the dependencies could be added to the plugin in order to ensure that the IDE contains the tools needed for JEE and Android development, in order to allow the developer to generate the DataAPI in Eclipse and use this code in other IDEs like Android Studio. Additionally, this allows the developer to import the server project in other IDEs.

- Android developing tools. This set of plugins extends Eclipse to be used as an IDE to develop Android applications.
- j2ee tools. This set of plugin provides facilities for building web applications. These facilities are needed to build the application server after configuring it.

Two extensions were created for the plugin. The New model wizard extension provides the developer with a wizard to create an emf file to define the data model using the Emfatic editor, and Generate Code popup menu extension triggers the code validation and generation modules.

The implementation of this extension is detailed below:

A. New Model Wizard Extension

This extension provides a wizard that allows the programmer to create a new SyncAndroid data model. As it can be seen in Figure 6-1, a new wizard is created in Eclipse which is located with other wizards that ADT provides for Android development. When the wizard starts, a form is displayed to the user in order to define the container project where the API will be generated and the name of the file where the data

1 http://www.couchbase.com/download#cb-mobile
model will be stored. After completing the form, a folder called “model” is created in the container project which contains an empty emf file that is automatically opened in the Emfatic editor.

The implemented configuration for this extension can be seen in Listing 6-1.

```xml
<extension point="org.eclipse.ui.newWizards">
  <wizard>
    <name>New SyncAndroid Data Model "
    <icon>icons/generateall.png"
    <category=com.android.ide.eclipse.wizards.category"
    <class=com.york.cs.swe.syncdoc.wizards.NewModelWizard"
    <id=com.york.cs.swe.syncdoc.wizards.NewModelWizard"
    <project="false">
    <description>
      Generate a new SyncAndroid data model
    </description>
  </wizard>
  <category id="com.android.ide.eclipse.wizards.category"
    <name="Android"/>
</category>
</extension>

Listing 6-1. New SyncAndroid Wizard Configuration
```

![Figure 6-1. New SyncAndroid Model Wizard](image)

Using the Emfatic language a data model for the application can be defined in the file created by the New SyncAndroid Wizard. The use of Emfatic language allowed to reduce implementation overhead, providing the developer with a way to represent a data model in a textual form using a simple language in a feature-rich text editor. Listing 6-2 shows a simple example of a data model represented by the DSL.
package com.york.cs.todolite2.document2;

@db
class TodoDB {
    val ListTasks[*] listTasks;
    val Task[*] tasks;
}

@document(sharing="shareable")
class ListTasks {
    attr String title;
    ref Task[*] tasks;
}

@document(sharing="shareable")
class Task{
    attr String title;
    attr boolean checked;
    @image
    attr String image;
    val Location[1] location;
    ref ListTasks[1]#tasks listTask;
}

class Location{
    attr String name;
    attr String address;
}

Listing 6-2. ToDo Model Example

Emfatic language also supports concepts like inheritance and enumeration types. More details about these and other concepts can be found in [106]. An important limitation of the DSL is that custom data types are not supported. Figure 6-2 shows a data model defined in an Android project using this editor. Also, it is important to notice that Emfatic editor validates rules defined for Ecore meta-model providing marks and error messages when errors are detected.
B. Generate Code Popup Menu Extension

This extension creates a contextual menu that can be used to generate the Java classes that constitute the DataAPI and the configuration files for the Android application and the server as shown in Figure 6-3.

![Image of Generate Code Popup Menu]

Figure 6-3. Generate Code Popup Menu

This popup menu calls SyncDocObjectActionDelegate class which implements the run method of IObjectActionDelegate interface. This method is triggered when the menu is selected and calls SyncDocGenerator.generate method. Finally, it refreshes the project at the end of the process in order to make the created files visible. During this process, Model Validation and Code Generator components are used.

```java
Context Eclass {
    // Not containment reference can have as target only document classes
    constraint DocumentReference {
        guard: not self.isDocument() and not self.isAnnotatedAs("db")
        check : self.areDocumentContRefOnlyToDocuments()
        message : "Not all containment reference from \""+self.name +\"" class are targeted to document classes"
    }
}
```

Listing 6-3. EVL Constraint example.

i. Model Validation Component

As explained in section 5.2.3.A, the first step in the code generation process is the validation of the model against the constraints defined in validator.evl program. Constraints listed in section 5.2.3.A were implemented using the Epsilon Validation Language (EVL). Listing 6-3 illustrates the implementation of DocumentReference constraint which verifies that non containment references in document classes have as target only document classes. As it can be seen in code of Listing 6-3, the constraint provides an error message that is presented to the user if the model is invalidated by the constraint in order to make it easy for the developer to identify the error and to correct it. Figure 6-4 shows how errors are displayed to the user through an Eclipse view.

![Image of Model Validation Error Messages]

Figure 6-4. Model Validation Error Messages

Listing 6-3 also shows that EOL operations are used in the EVL program. A wide variety of operations were defined to be used in the validation program and the generation templates. These operations were implemented in two different programs: ECoreUtils.eol and JavaUtils.eol using the Epsilon Object Languages (EOL).
ii. Code Generator Component

This component is triggered by the Generate Code Popup Menu extension when the model validator has not detected errors in the data model. As explained in section 5.2.3.A.iii, this component generates the code of the DataAPI and configuration files using the templates defined in Figure 5-6. The execution of these templates is coordinated by two different programs in order to separate the generation of Java code and configuration files. Listing 6-4 presents one of the rules used to coordinate the execution of EGL programs. As it can be seen in this code, the coordination programs filter the elements that are sent to EGL programs using operations defined in EOL programs.

```
import "javautil.eol";
import "ecoreutil.eol";
...
rule EClass2Class
    transform c : EClass {
        guard : not c.isAnnotatedAs("db")
        template : "eclass2class.egl"
        target : c.name + ".java"
    }
...
```

Listing 6-4. Coordination Rules of EGL Templates.

EGL programs uses templates for each type of the classes detailed in section 5.2.3.A.iii. Listing 6-5 illustrates a section of code of the eclass2class.egl program used to generate setter methods for references between document classes. The second part of these codes present the code generated by the EGL program. This example illustrates the generation of the reference listTask defined in the model presented in Listing 5-1. A complete class diagram of the generated code of this model can be seen in Appendix B-1 Examples of the templates used to generate configuration files can be seen in Appendix C-3 and a configuration file example is available in Appendix C-4.

```
[%for (r in c.eReferences.select(r| not r.isMany and r.eReferenceType.isDocument())) %]
    public [%=c.name%] set[%=r.name.ftuc()%][%=r.eType.getJavaName()%] [%=r.name%]) {
        if (this.[%=r.name%] != [%=r.name%]) {
            if ([%=r.name%] == null) {
                dbObject.put("[%=r.name%]", new HashMap<String, Object>());
            }
            if (this.[%=r.name%] !=null) this.[%=r.name%] .removeReferencedBy(getId());
        } else {
            createReference("[%=r.name%]", [%=r.name%]);
        }
        this.[%=r.name%] = [%=r.name%];
        notifyChanged();
    }
    return this;
[%]}

public Task setListTask(ListTasks listTask) {
    if (this.listTask != listTask) {
        if (listTask == null) {
            dbObject.put("listTask", new HashMap<String, Object>());
            if(this.listTask !=null) this.listTask .removeReferencedBy(getId());
        } else {
            createReference("listTask", listTask);
        }
        this.listTask = listTask;
        notifyChanged();
    }
```

Listing 6-6. Generation Template for Setter Method of References

After code generation, the developer can find the files generated for the DataAPI component in the folder “src-gen” (1) as shown in Figure 6-5. This folder is created in the Android application project and it is configured as a source path of the project. Configuration files are located in a separate folder called “configurationFiles” (2) with a readme file where instructions to configure the system are provided.

Figure 6-5. Files generated by Code Generator

6.2.2. SyncAPI Component

This component contains the subcomponents DataAPI, Object Relational Mapper API, User Manager API and the Synchronization Manager API as illustrated in Figure 5-3. The first subcomponent component is generated by the Code Generator component, as explained before. The three last components are contained in an Android Library Project\(^1\) called SyncDocALib. This project must be imported into the workspace and should be referenced as a library in the project that will use it. Delivering these components as an Android library project makes it possible to use not only Java classes on them, but also use resources like XML files.

A. SyncData API and Data Relational Mapper

These components allow the developer to interact with the database. The structure of these components and the services that they provide are detailed in section 5.2.3.B. This section will contain an explanation of how the developer can use them. Listing 6-6 illustrates how the Data API generated from the

\(^1\) Android Library Projects contain shareable Android source code and resources that you can be referenced in Android projects. Library projects cannot be installed onto a device, they are pulled into the application file at build time.[126]
data model provided in Listing 5-1 can be used. A detailed class diagram of the API generated for this model is presented in Appendix B-1.

```java
Application application = (Application) getApplicationContext();
TodoDB todoDB = application.getTodoDB();
ListTasks listTasks = new ListTasks();
Task task1 = new Task();
Task task2 = new Task();
listTasks.setTitle("listtitle");
task1.setTitle("task1");
Bitmap.Config conf = Bitmap.Config.ARGB_8888;
Bitmap image = Bitmap.createBitmap(w, h, conf);
task1.setImage(image);
task2.setTitle("task2");
listTasks.getTasks().add(task1);
listTasks.getTasks().add(task2);
todoDB.getListTasks().add(listTasks);
todoDB.getTasks().add(task1);
todoDB.getTasks().add(task2);
todoDB.sync(true);
List<Task> tasksFromDBs = todoDB.getTasks().findByKey(Task.Task_title, 0, 0, "task1");
Task taskFromDB = taskFromDBs.getIterator.next();
```

Listing 6-6. DataAPI Use

In lines 01 and 05, the database TodoDB instance is retrieved from the context of the application since it was initiated when the application started. Then, between 07 to 21 a ListTask and two List objects are created setting their attributes. Line 19 illustrates how a bitmap object is attached to one of the tasks objects. References between the ListTask object and the Task objects are set in lines 23 and 24. From line 26 to line 28, the created objects are added to the database. While all these operations are performed, all objects are kept in the cache but no synchronization is made with the database. The data is persisted in database only when the “sync” method of the TodoDB object is called in line 30. It is recommended that sync operation is not called from the UI thread\(^1\) in order to keep the application responsive as it also should be done with other databases like SQLite.

Finally, line 32 illustrates how a list of tasks is retrieved from the database using as parameter the title attribute. It is recommended that this method is not called from the UI thread because it can take more time depending in the complexity of the query performed.

\(^1\) UI thread (user interface thread) should not perform long running processes. Different methods can be used to avoid the use of the UI thread as can be seen in [127].
As shown in the example, DataAPI abstracts the difficulties of using NoSQL databases and the management of metadata used for data synchronization. The services provided by this API are based on the component Object-Relation Mapper, which is invisible for the developer.

### B. User Manager API

The user manager component abstracts user account management from the developer providing an API that allows to manage accounts without worrying about complex interactions with Android services, the application server or authentication tokens. Since the structure and services provided by this API are detailed in section 5.2.3.B.iii, this section illustrates how Android developers can use it.

Listing 6-7 illustrates how this API can be used. From line 01 to line 04 a user object is created and its attributes are set. Using this user object and the method provided by UserManager class, a new user account can be created or an existing user can be authenticated as shown in lines 6 and 7. In line 8 the current account is removed, and in line 9 the API is used to verify if an account was already created in the application. Moreover, data of the user can be retrieved as illustrated in line 10. It is recommended that these operations are not called from the UI thread\(^1\) in order to keep the application responsive as this operation is performed as its duration can vary depending on the quality of the network connection.

```java
01 User user = new User();
02 user.setName("usertestFromApp Name2");
03 user.setEmail("usertestFromApp@email2");
04 user.setPassword("usertestFromApppassword2");
05
06 UserManager.userSignUp(getContext(), user);
07 UserManager.userSignIn(getContext(), user);
08 UserManager.removeCurretUserFromApp(context);
09 boolean existsAcc=UserManager.existsAccount(getContext());
10 UserManager.getCurrentUserEmail(getContext());
```

Listing 6-7. User Manager API Usage Example

Additionally, User Manager API provides developer activities\(^2\) or user interfaces that can be used to register new accounts and login accounts. These activities are defined in the SyncADocLibrary, which provides a class which allows to create menus to call these activities from activities created by the developer. Figure 6-6 illustrates the activity that the library provides.

![Sing Up]

Figure 6-6. User Registration Activity.

---

1. UI thread (user interface thread) should not perform long running processes. Different methods can be used to avoid the use of the UI thread as can be seen in [127].
2. An Activity is an application component which provides a user interface with which users can interact in order to do something. [128]
To use these activities the developer shall extend the class BaseActivity which is an extension of the Android Activity class. Then the developer can add buttons to the menu using the methods illustrated in Listing 6-8.

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    if (!mDrawerFragment.isDrawerOpen()) {
        getMenuInflater().inflate(R.menu.main, menu);
        // Add Login button if the user has not been logged in.
        addLoginButton(menu);
        // Add Share button if the user has been logged in
        addShareButton(menu);
        // Add synchronization button if the user has been logged in
        addSyncNowButton(menu);
        // Add remove existing user account
        addRemoveaccountNowButton(menu);
        restoreActionBar();
        return true;
    }
    return super.onCreateOptionsMenu(menu);
}
```

Listing 6-8. Use of Activities Provided by SyncADocLib Library

Finally, it is worth noticing that this component needs a configuration file to work. This configuration file is produced by the code generator component with the name “authenticator.xml”, which should be copied into the xml folder of the resources (“res”) folder of the Android project. An example of this file generated for the TODO data model presented in Listing 6-2 can be seen in Appendix C-6. Furthermore, permissions, services and activities must be declared in the Manifest file in order to use this API. The needed code for this configuration is also produced by the system. An example of this code can be seen in Appendix C-5

C. Synchronization Manager API

The synchronization manager component abstracts the management of the synchronization process from the developer providing an API that allows to schedule and trigger this process without worrying about complex interactions with Android services, the application server or authentication processes. Since the structure and services provided by this API are detailed in section 5.2.3.B.iv, this section illustrates how Android developers can use it.

As it can be seen in Listing 6-9, this API provides methods to perform several operations over the synchronization process. SyncCurrentUserNow method starts the synchronization process immediately while syncScheduledPeriodically schedules the process to be triggered periodically. syncWithNetMsg sets the synchronization process to be triggered automatically when a network connection is available. Finally, isSyncActive and isSyncPending methods allows to know if a synchronization process is running and if a synchronization process is pending respectively.

```java
SyncManager.syncCurretUserNow(getContext());
SyncManager.syncScheduledPeriodically(context,(long) (60*60*2));
SyncManager.syncWithNetMsg(getContext());
SyncManager.unscheduleSync(getContext());
SyncManager.isSyncActive(getContext());
SyncManager.isSyncPending(getContext());
```

Listing 6-9. Use of Synchronization Manager API
It is worth noticing that this component needs a configuration file to work. This configuration file is produced by the code generator component with the name “sync_adapter.xml”, which should be copied into the xml folder of the resources (“res”) folder of the Android project. An example of this file generated for the TODO data model presented in Listing 6-2 can be seen in Appendix C-7. Moreover, permissions and services must be declared in the Manifest file in order to use this API. The needed code for this configuration is also produced by the system. An example of this code can be seen in Appendix C-5.

Finally, a menu item that triggers the synchronization process can be added to any activity using the BaseActivity class as illustrated in Listing 6-8 in line 11.

6.2.3. Central Repository

As explained in section 5.2.3.C, the server side of the system provides a central repository where Android applications can back up their data. It uses the facilities provided by the Couchbase Lite framework to provide this service. As it can be seen in Figure 5-3, this component has three subcomponents: the application server which provides authentication and authorization services, the SyncGateway server which controls the data synchronization process, and the Couchbase server which provides a NoSQL document database. This section details the implementation of these components.

A. Application Server

As explained in section 5.2.3.C.i, this component provides services for user creation and authorization using web services. This component was implemented using the Spring which provides facilities to develop web services in an annotation based Model-View-Controller (MVC) framework (Figure 6-7). Using annotations, controllers, handler mappers, exceptions and other resources, web applications can be easily developed to provide Restful web services avoiding the boilerplate for data representation, data mapping and web configuration. Restful web services provide stateless web services and bandwidth efficiency avoiding complexities of SOAP web services like the necessity of formal contracts [129]. It should also be noted that Couchbase tools use Restful web services. Therefore their use in the authentication services allows to keep consistency across the system.

The application server component contains one controller where one web service is defined to process all requests for users’ authentication using a request handler. An example of the code to define a request handler is illustrated in Listing 6-10. As it can be seen in this code, the web service is exposed under the URL “/User” and accepts post calls. As a parameter an object of the User class must be delivered.

---

1 Spring is an open source application framework and inversion of control container for the Java platform providing extensions for building web applications on top of the Java EE platform.
Listing 6-10. Web Service Handler Definition

Listing 6-10 also shows that the response of the web service contains a SessionSync object where the authorization information is delivered. SessionSync and User objects are included in the http body of http messages as JSON representations. User and SessionSync classes are also defined in the Android application to parse the messages. Additionally, different exceptions are defined for the web services providing messages and different HTTP codes to allow the developer to detect errors easily as defined in the web service design in Figure 5-13. Listing 6-11 illustrates the definition of the exception which is raised when a request with incorrect user credentials is received by the server.


As shown in Figure 5-3, the server application also uses web services to manage user accounts in SyncGateway. Listing 6-12 illustrates how this interaction was implemented. The method userExists confirms if an account with a specified email exists in SyncGateway.

Listing 6-12.

```java
public static boolean userExists(String email) throws Exception {
    URL url;
    try {
        url = new URL(AuthenticatorServiceImpl.SERVER_URI_SYNC_USER + email);
    }

    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod("GET");
    connection.setDoOutput(true);
    int responseCode = connection.getResponseCode();
    if (responseCode == HttpStatus.OK.value()) {
        return true;
    } else if (responseCode == HttpStatus.NOT_FOUND.value()) {
        return false;
    } else {
        throw new Exception("Not possible to validate User");
    }
}
```
Finally, it is important to notice that this component does not depend on the model defined for the application. This is why the developer only needs to configure the IP address of the SyncGateway and the name of the database in the Couchbase server before deploying the application in a Tomcat server. These addresses must be specified in class AuthenticatorServiceImpl as illustrated in Listing 6-13.

```java
public class AuthenticatorServiceImpl implements AuthenticatorService {
    public static final String SERVER_URI_SYNC_ADMIN = "http://localhost:4985";
    public static final String SERVER_URI_SYNC_NOADMIN = "http://localhost:4984";
    public static final String SYNC_DATABASE_NAME = "todos";
}
```

Listing 6-13. Application Server Configuration

B. SyncGateway and Couchbase Server.

As it was explained in section 2.3.3, SyncGateway and Couchbase server provide the services of data synchronization and database server respectively. Since these components were not implemented in this project, this section briefly explains their configuration.

As explained in section 6.1.2.B.ii, a configuration file called “config.json” is generated by the system. This file contains the synchronization rules generated from the data model (an example of this file can be found in Appendix C-4). To run SyncGateway, this file should be used as a parameter when the server is started with the command sync_gateway as can be seen in Figure 6-8.

![Figure 6-8. SyncGateway Start Command.](http://www.couchbase.com/download#cb-mobile)

After installing Couchbase server, a database (called bucket in Couchbase) should be created using the administration console. This administration console can be accessed using a web server and provides wizards for creating databases as illustrated in Figure 6-9.

![Figure 6-9. Couchbase Server Console](http://www.couchbase.com/download#cb-mobile)

1 [http://www.couchbase.com/download#cb-mobile](http://www.couchbase.com/download#cb-mobile)
Finally, the addresses of the application server and SyncGateway should be configured in the Android application in class Application as shown in Listing 6-14.

```java
/* set here your app configuration */
/* url of the tomcat server where syncserver will be deployed */
SERVER_URL = "http://10.0.3.2:8080/SyncService";
/* url of the server where syncGateway will be deployed */
GATEWAY_URL = "http://10.0.3.2:4985";
ROOT_PACKAGE = "com.york.cs.todolite2.document2";
/* local database name, */
DATABASE_NAME = "todos";
```


6.3. Summary

This chapter describes the implementation of the system. First, a summary of the preparation of the developing environment is discussed. Then, the implementation of each component is presented along with an explanation of how they can be used by Android developers.

The following list summaries the steps that Android developers should follow to use the system:

1. To install the SyncDoc plugin in Eclipse along with all its dependencies (ADT [134], Web Tools Platform [81], Epsilon [135]). It is recommended to install Eclipse Kepler IDE for Java EE Developer version.
2. To prepare the development environment. Couchbase server and SyncGateway should be installed in order to run server side of the system.
3. To import to the workspace the SyncDocALib Android library and the application server projects.
4. To create an Android application using the Android Development Tools (ADT) plugin.
5. To reference the SyncDocALib library project as a library.
6. To create a new data model using the SyncAndroid plugin.
7. To generate the DataAPI and configurations files using the SyncAndroid plugin. If the data model contains errors, they should be corrected following the instruction provided by the plugin before attempting to generate the code again. The Data API is created in source file “src-gen” and configuration files in folder “/model/configurationFiles” in the Android project.
8. To configure server address in Application class of the DataAPI. “Authenticator.xml” and “sync_adapter.xml” configurations files should be copied in folder “/res/xml/” in the Android project.
9. To configure the IP addresses of SyncGateway in the application server in class “AuthenticatorServiceImpl”.
10. Before running the application, the application server should be deployed in a Tomcat server, SyncGateway should be started with the configuration file “config.json” and Couchbase server should contain a database to store application data.
Chapter 7
EVALUATION

The SyncAndroid system has been evaluated using two approaches. First, integration testing was performed on the system where the correctness of the code implementation was evaluated. Then, the usefulness of the system was evaluated using a case study. These evaluations are presented in this chapter.

7.1. Testing

This section describes the testing process performed on the system. First, the approach used to test the application is discussed. Then, most relevant test cases used in the process and their results are described.

7.1.1. Testing Approach

The approach used to test the system was adapted to the iterative nature of the development methodology used in this project. In each iteration of the implementation phase, a Bottom Up approach was used, where the test was conducted from sub components to main components. Each of the components defined in the high level architecture (Figure 5-3) were tested individually when they were implemented in order to detect errors in each of them. The purpose of this exercise is to detect and identify as many errors as possible before the integration of the components since the test conditions are easier to create and the test results are easier to observe in individual components. After each component completed the test phase, they were integrated to the system and a new test process was conducted.

Also, it is worth noticing that automated testing was used when possible. The use of automated testing allowed to run test cases every time an important change was made in the system. Nevertheless, not all test cases could be automated because of the nature of the system. Some of the test cases involved the use of more than one Android device performing different operations in a specific order. Additionally, it should be noted that some of the security strategies implemented in Couchbase Tools made the automation of test difficult to achieve. For example, information stored in the Couchbase server can only be accessed from the same machine where it is located. Therefore, test cases which run in Android emulators could not interact directly with the Couchbase database server.

To ensure test results, testing was performed in each iteration of the implementation process resulting in a big collection of test cases. Due to space limitation, only the most relevant test cases are documented in the next section. This documentation is based in the IEEE Standard 829-2008 for Software and System Test Documentation [136] but is summarized to the most relevant information.

7.1.2. Test Case specification and Logs.

Most relevant test cases and their result are detailed below:

<table>
<thead>
<tr>
<th>Test Case 1</th>
<th>: SA-CD-VL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Items:</strong> Code Generator Component: Model Editor, Model Validator and Code Generator</td>
<td></td>
</tr>
<tr>
<td><strong>Test Procedure:</strong></td>
<td></td>
</tr>
<tr>
<td>1. Create a new Android project</td>
<td></td>
</tr>
<tr>
<td>2. Create a new model containing errors. (Examples of errors can be seen in Appendix D-1)</td>
<td></td>
</tr>
<tr>
<td>3. Run Code Generator</td>
<td></td>
</tr>
<tr>
<td><strong>Output Specification:</strong> Code generator interrupts code generation process, displays an error dialog asking for corrections and shows error messages in SyncAndroid View.</td>
<td></td>
</tr>
<tr>
<td><strong>Type of Test:</strong> Manual test</td>
<td></td>
</tr>
<tr>
<td><strong>Actual Output:</strong> Code generator interrupted code generation process, displayed an error dialog asking for corrections and showed error messages in SyncAndroid View.</td>
<td></td>
</tr>
<tr>
<td><strong>Result:</strong> Pass</td>
<td></td>
</tr>
</tbody>
</table>
**Notes:** The test was performed with several data models, an example of one of the models used can be seen in Appendix D-1

**Test Case 2**  : SA-CD-FG  
**Test Items:** Code Generator Component: Model Editor, Model Validator, Code Generator  
**Test Procedure:**  
1. Create a new Android project  
2. Create a new model.  
3. Run Code Generator  
**Output Specification:** Code generator creates “src-gen” folder and adds it to project as source folder. Valid DataAPI is generated in “src-gen”. Valid configuration files are generated in “/model/configurationFiles” folder.  
**Type of Test:** Manual test  
**Actual Output:** DataAPI generated correctly. Configuration files generated correctly but server configuration file (config.json) was generated with Windows line delimiter which causes errors in SyncGateway.  
**Result:** Partial Pass  
**Notes:** Text line delimiters are different for UNIX and Windows [131] platforms and server’s file configuration must use UNIX line delimiters. File encoding depends on Eclipse configuration. It can be set in “Window > Preferences > General > Workspace > New text file line delimiter” before code generation or external tools like Notepad ++ can be used to change this codification (Notepad ++: “Edit > EOL conversion>UNIX / OSX format”).

**Test Case 3**  : SA-CD-AU1  
**Test Items:** UserManagerAPI, Application Server  
**Environmental Needs:** Project configured with model “Blog”. (Model can be seen in Appendix C-2.)  
**Test Procedure:**  
1. Create a new user  
2. Call Authentication service without user mail and name parameters.  
**Output Specification:** Authentication services must raise “AuthenticationExceptionServer” with status “HttpStatus.SC_BAD_REQUEST”.  
**Type of Test:** Automatic test (test case: Class: TestAppAuthenticator, method: testAuthenticationIncorrectInformation.)  
**Actual Output:** Authentication service raised “AuthenticationExceptionServer” exception with status “HttpStatus.SC_BAD_REQUEST”.  
**Result:** Pass

**Test Case 4**  : SA-CD-AU2  
**Test Items:** UserManagerAPI, Application Server  
**Environmental Needs:** Project configured with model “Blog”. (Model can be seen in Appendix C-2.)  
**Test Procedure:**  
1. Create a new user  
2. Call Authentication sign up service.  
**Output Specification:** Authentication token must be returned. New user must be created in application server database.  
**Type of Test:** semi-automated test (test case: Class: TestAppAuthenticator, method: testSignUpNewUser.)  
**Actual Output:** Authentication token was returned and user was created in application server database.  
**Result:** Pass

**Test Case 5**  : SA-CD-AU2  
**Test Items:** UserManagerAPI, Application Server  
**Environmental Needs:** Project configured with model “Blog”. (Model can be seen in Appendix C-2.)  
**Test Procedure:**  
1. Create a new user  
2. Call Authentication sign up service.  
**Output Specification:** Authentication token must be returned. New user must be created in application server database.
| Test Case 6  | SA-CD-AU2 |
| Test Items: | UserManagerAPI, Application Server |
| Environmental Needs: | Project configured with model “Blog”. (Model can be seen in Appendix C-2.) |
| Test Procedure: | 1. Create data in the database using the API  
2. Create a new user  
3. Remove current user from application  
4. Create new data and persist in database  
5. Sign Up new user using API  
6. Remove current user |
| Output Specification: | 1. After step 1 owner field must be null in all documents in the database.  
2. After step 2 owner field must be equal to the user’s email in all documents in the database.  
3. After step 3 owner field must be null in all documents in the database.  
4. After step 4 owner field must be null in all documents in the database.  
5. After step 5 profile document must be available for the new user.  
6. After step 5 owner field in all documents in database must be the new user’s email.  
7. After step 7 assert that all documents were purged from the database. |

| Test Case 7  | SA-CD-SYNC1 |
| Test Items: | SyncManagerAPI, Application Server |
| Environmental Needs: | Project configured with model “Blog”. (Model can be seen in Appendix C-2.). User registered in application before running the test. |
| Test Procedure: | 1. Trigger synchronization process.  
2. Schedule periodic synchronization every 2 minutes.  
3. Disconnect network in device.  
4. Set synchronization trigger in network mode  
5. Connect network in device. |
| Output Specification: | 1. After step 1 synchronization process must start in SyncGateway console.  
2. After step 2 synchronization process must be triggered in SyncGateway console in 2 minutes periods approximately.  
3. After step 5 it is verified that synchronization process is triggered in SyncGateway console. |

| Test Case 8  | SA-CD-D5 |
| Test Items: | DataAPI, Object Relation Mapper |
| Environmental Needs: | Project configured with model “Blog”. (Model can be seen in Appendix C-2.). |
2. Populate database using the DataAPI containing nested objects and relations between documents  
3. Persist database with option to clear cache set to true |
| Output Specification: | 1. Database documents can be retrieved directly from database using objects’ identification numbers (id). |
2. Retrieve objects using method “findById” of DataAPI collection and verify that data is consistent.
3. Retrieve objects using a query producer and method “findByKey” of DataAPI collection and verify that data is consistent using an Iterator.
4. Retrieve an object through a reference verify that data is consistent using an Iterator.
5. Update data in objects and verify that data is updated in database retrieving documents directly from it.
6. Delete objects using DataAPI and verify that documents are deleted in database retrieving documents directly from it.

**Type of Test:** Automated test (synchronization set using test class: BlogTestAPI )

**Actual Output:** All verifications were correct

**Result:** Pass

For the following test cases, this environment specifications are common:
- The test was conducted using the application TodoLite2. This application runs over the SyncAndroid library, with an API produced using the data model ToDo presented in Appendix D-3And configuration files presented in 0, 0, 0, 0and Appendix D-9
- The test environment was configured following the instructions presented in section 6.3.
- Genymotion was used to emulate 3 Android devices. These emulated devices are references as device1, device2 and device 3 in test cases.
- User1 and user2 refer to two different user accounts.
- ToDo application was configured to clean the cache after each synchronization in order to present chances immediately. This was done in the application class setting option “CLEAR_CHACHE_AFTER_SYNC=true;”
- The database in central repository was cleaned before testing.
- Before each test the network connection is disabled in all devices, before triggering synchronization it is enabled and after finishing synchronization it is disabled immediately.
- Test cases were conducted in the order presented here.
- Tested items are all modules that comprise the system and their integration in an Android application.

**Test Case 9 : SA-CD-SYNC20**

**Test Procedure:**
1. In device 1:
   a. Register user 1.
   b. Create 2 lists with 2 task2 in each one.
   c. Trigger synchronization
2. In device 2:
   a. Log in with user 1.
   b. Trigger synchronization.
3. In device 3:
   a. Log in with user 2.
   b. Trigger synchronization.

**Output Specification:**
1. Device 1 and 2 should contain the same information after test procedure.
2. Device 3 should display any information.

**Type of Test:** Manual test

**Actual Output:** Device 1 and 2 contains the same information after test procedure. Device 3 did not synchronize data of devices 1 or 2.

**Result:** Pass

**Test Case 10 : SA-CD-SYNC21**

**Test Procedure:**
1. In device 2 with user1 account:
   a. Add one task to list1.
   b. Trigger synchronization
2. In device 1 with user1 account:
   a. Trigger synchronization.
3. In device 3 with user2 account:
   a. Trigger synchronization.

**Output Specification:**
1. Device 1 and 2 should contain the same information after test procedure.
2. Device 3 should not contain any information.

**Type of Test:** Manual test

**Actual Output: Actual Output:** Device 1 and 2 displayed the same information after test procedure. Device 3 did not synchronize data of devices 1 or 2.

**Result:** Pass

### Test Case 11 : SA-CD-SYNC22

**Test Procedure:**
1. In device 1 with user1 account:
   a. Add one task to list1.
2. In device 2 with user1 account:
   a. Delete list1
3. Trigger synchronization process in the following order: device 1, device 2, device 1, and device 2. The synchronization should be triggered twice in each device in order to solve conflicts
4. In device 3 with user2 account:
   a. Trigger synchronization.

**Output Specification:**
1. Device 1 and 2 should contain the same information after test procedure. Deleted list, list 1, should be available again in device 1 and 2.
2. Device 3 should not contain any information.

**Type of Test:** Manual test

**Actual Output: Actual Output:** Device 1 and 2 contain the same information after test procedure. Deleted list, list 1, is available again in device 1 and 2. Device 3 does not contain any information.

**Result:** Pass

### Test Case 12 : SA-CD-SYNC23

**Test Procedure:**
1. In device 1 with user1 account:
   a. Edit one task of list1.
2. In device 2 with user1 account:
   a. Edit the same task of list1.
3. Trigger synchronization process in the following order: device 1, device 2, device 1, and device 2. The synchronization should be triggered twice in each device in order to solve conflicts
4. In device 3 with user2 account:
   a. Trigger synchronization.

**Output Specification:**
1. Device 1 and 2 should contain the same information after test procedure. Conflicts should be resolved by the API.
2. Device 3 should not contain any information.

**Type of Test:** Manual test

**Actual Output: Actual Output:** Device 1 and 2 contain the same information after test procedure. Conflicts were resolved by the API. Device 3 does not contain any information.

**Result:** Pass

### Test Case 13 : SA-CD-SYNC24

**Test Procedure:**
1. In device 1 with user1 account:
   a. Add image to one task of list1.
   b. Trigger synchronization.
2. In device 2 with user1 account:
   a. Trigger synchronization.
3. In device 3 with user2 account:
7.1.3. Test Summary

The tests conducted during iterations were performed successfully. Test cases conducted in each development iteration allowed to detect most of the bugs of the system at an early stage making it easier to correct these bugs. It is worth noticing that during the iterations test, the SyncAPI component which runs in Android and the Central repository server which runs in a web server (Tomcat 7) were tested separately in order to ensure their correctness before integrating them. This also allowed to build automated test cases which were used in each iteration and after important changes were made in the system.

To test the entire product, an application was developed using the system in order to not only test the correctness of the code but also to test the integration of the system in an Android application. This was very useful because it allowed to detect not only code bugs, but also flaws of the design of the interfaces provided by the API. In some cases these interfaces were redesigned to make them easier to use for
developers. Almost all bugs were corrected after the test process with the exceptions that are documented in the test case log.

Several Java coverage tools were tested to monitor the code coverage of the test, although it was not possible to get a coverage report from any of these tools. Popular free Java code coverage tools like EclEmma and Emma do not provide official support for Android code as it can be seen in their official documentation [138]–[140]. Several ways to adapt these tools to monitor Android code were found on the Internet [141]–[144], but all of them monitor the code in the tested project and none of them could be used to monitor an Android library project. As most of the components of the SyncAPI reside in an Android library project, no test coverage report could be produced.

The lack of coverage monitoring was a problem difficult to solve in a system that contains 7158 instructions and 431 methods contained in 70 classes. In order to counteract this problem, there was special attention paid to component testing, where methods are easier to test and it is easier to analyse the coverage of test cases. In contrast, this is very difficult to achieve in integration testing. Also, special attention was given to produce several test cases with different inputs.

7.2. Case Study

In order to evaluate the SyncAndroid system, a demonstrator application was developed. This application was developed in two different versions, one using the SyncAndroid System and a second version without using it. This exercise provided the opportunity to evaluate the usefulness of the system and the fulfilment of some of the non-functional requirements.

7.2.1. Evaluation Criteria

This case study will evaluate the usefulness of the SyncAndroid. This case study focuses its attention on the evaluation of these non-functional requirements:

- The system shall provide an easy to learn and memorize API. (SY-NFR-01)
- The system shall provide the developer team manager with facilities to successfully deliver applications on time and with less effort. (SY-NFR-06)
- The time to perform operations over the database shall not be severely impacted by the system. (SY-NFR-05)

7.2.2. Demonstrator Application

To demonstrate the benefits provided by the SyncAndroid system, the demonstration application “ToDo Lite Sync” of Couchbase Lite products was used as a base. This application is available in Google Play[146] and its source code is available under the Apache Licence 2.0 [147]. The use of a predefined application prevented from implementing a demonstrator application directed to make easier to use SyncAndroid system to fulfil the evaluation criteria and also reduced the time required for its implementation, especially in the definition of requirement and design stages.

ToDo Lite Sync allows to manage multiple to-do lists and share them with other users. It also allows to associate photographs with tasks. The information contained in this app is synchronized “in the cloud” using a non-relation NoSQL database [147]. This application was enhanced to provide user authentication services using Android accounts and synchronization management using the facilities of syncAdapter. (This was discussed in section 5.2.3.B, since the original version of the application uses Facebook authentication and only provides synchronization on demand.

---

1 Google Play is the official application store for Android. Todo Lite Sync applications also is available for IOS in iTunes. [145]
The same application was developed using the SyncAndroid system providing exactly the same functionalities as the original application in order to make a comparison between both applications and the development experience in both cases.

Also it is important to notice that the same central repository server was used for both cases since both applications use the same authentication system. Nevertheless, the configuration file for SyncGateway produced by the system is different from the manually coded ones in order to use two different databases in the server.

![Diagram of ToDo Lite Design](image)

*Figure 7-1.ToDo Lite Design*

Figure 7-1 illustrates a high level design. This design is shared by both applications, but the application which was developed using the SyncAndroid system uses the facilities provided by the system to implement all the components that are inside the green boxes in the diagram, while in the other application these components were manually coded. Some snapshots of the applications can be seen in Appendix D-10 and Appendix D-11.

7.2.3. Usability

As claimed in requirement SY-NFR-01, one of the goals of the system was to provide an easy to learn, memorize and use API. Additionally, requirement SY-NFR-06 shall provide the developer team manager with facilities to successfully deliver applications on time and with less effort. This section analyses how an Android developer can implement the functionalities of the application using the SyncAndroid system and manually coding all components.

A. Database interaction

For the hand coded version of ToDo application, the Android API provided by Couchbase Lite was used while in the version which uses SyncAndroid’s facilities, the DataAPI was used. Table 7-1 compares the implementation of both versions.
<table>
<thead>
<tr>
<th>No</th>
<th>Number of lines of code needed for implementation of activities</th>
<th>ToDo Lite Hand coded version</th>
<th>ToDo Lite Using SyncAndroid system</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2862</td>
<td>2369</td>
</tr>
<tr>
<td>2</td>
<td>Knowledge required for implementation</td>
<td>NoSQL databases concepts</td>
<td>Emfatic language to define the meta model (see section A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Map-reduce concepts used for querying database.</td>
<td>Data API (see section A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Couchbase Lite API</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Synchronization metadata used in Couchbase Lite tools.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Advantages</td>
<td>A more efficient code can be achieved to perform CRUD operation according to the particular requirements of the application.</td>
<td>Fast and easy to implement (see item 1 in this table)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The developer does not need to know how to use complex NoSQL concepts (see item 2 in this table).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Model classes and configuration files are produced by the system automatically.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NoSQL querying logic and database document structure are abstracted from developer</td>
</tr>
<tr>
<td>4</td>
<td>Disadvantages</td>
<td>Time and effort needed for implementation is bigger.</td>
<td>SyncAndroid does not support aggregation queries.¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developer should know how to use items listed in row 2 in this table.</td>
<td></td>
</tr>
</tbody>
</table>

Table 7-1. ToDo Lite-Database Interaction. Hand coded vs SyncAndroid.

Table 7-1 compares the implementation of database CRUD operations using the SyncAndroid system and hand coding them. Nevertheless, it is important to notice that most of the code referenced in row 1 of the table is used to implement the user interface logic, therefore it is necessary to detail how the developer interacts with the database.

```java
01 //database document structure
02 Map<String, Object> properties = new HashMap<String, Object>();
03 properties.put("type", DOC_TYPE);
04     properties.put("title", title);
05     properties.put("checked", Boolean.FALSE);
06     properties.put("list_id", listId);
07  Document document = database.createDocument();
08  UnsavedRevision revision = document.createRevision();
09   revision.setUserProperties(properties);
10  if (image != null) {
11      ByteArrayOutputStream out = new ByteArrayOutputStream();
12      image.compress(Bitmap.CompressFormat.JPEG, 50, out);
13      ByteArrayInputStream in = new ByteArrayInputStream(out
```

¹ The group level aggregation query allows to collapse together (aggregate) rows with the same attributes and aggregated statistics of the grouped-together rows by using a reduced function that can be calculated.[106]
out.toByteArray();
revision.setAttachment("image", "image/jpg", in);
}
revision.save();

//update version of list which contains the task
Document list = database.getExistingDocument(listId);
if (list != null) {
    list.update(new Document.DocumentUpdater() {
        @Override
        public boolean update(UnsavedRevision newRevision) {
            Map<String, Object> properties = newRevision
                    .getUserProperties();
            newRevision.setUserProperties(properties);
            return true;
        }
    });
}

Listing 7-1. Hand Coded Task Creation

Task task = new Task();
task.setTitle(inputText);
getTodoDB().getTasks().add(task);
listTasks.getTasks().add(task);
if (mImageToBeAttached != null)
    task.setImage(mImageToBeAttached);
getTodoDB().sync();

Listing 7-2. Task Creation Using Data API

Listing 7-1 and Listing 7-2 illustrates the code necessary to create a new Task in the application both
not using and using the DataAPI. It clearly illustrates that the effort needed to code the same operation using
the DataAPI of the SyncAndroid system is dramatically reduced. Additionally, it is important to note that the
hand coded version uses Java maps to build the database document which can be even more difficult when
more complex objects are manipulated. Additionally, as Couchbase Lite does not provide support for
reference among database documents, it should be manually controlled as well as data type conversions.
This is a very simple example, when complex operations such as queries have to be coded, the difference is
even bigger.

B. Authentication and Synchronization Services

For the hand coded version of ToDo application, the authentication and synchronization services
were hand coded while in the application that used the SyncAndroid system, the SyncManager API and the
UserManager API were used to implement these services.

Table 7-2 compares the implementation of these services using the SyncAndroid system and hand
coding them:

<table>
<thead>
<tr>
<th>No</th>
<th>ToDo Lite Hand coded version</th>
<th>ToDo Lite Using SyncAndroid system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Synchronization Service</td>
<td>Authentication Service</td>
</tr>
<tr>
<td>1</td>
<td>Number of lines of code needed for implementation</td>
<td>977</td>
</tr>
<tr>
<td></td>
<td>Knowledge required for implementation</td>
<td>Android Account Management</td>
</tr>
<tr>
<td>---</td>
<td>--------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Android SyncAdapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Android services</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Content Resolvers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Couchbase Lite tools</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Web services</td>
</tr>
</tbody>
</table>

|   | Advantages                           | User attributes can be customized | Fast and easy to implement (see item 1 in this table) |
|   |                                      | Other types of authentication can be used (e.g. Facebook, Google Authorization) | The developer does not need to know how to use complex Android concepts (see item 2 in this table). |
|   |                                      | Authentication user interfaces can be customized. | Authentication server ready for user. |
|   |                                      |                                                          | API not only provides the services, also if it is not necessary to produce customized user interfaces for login, the developer can use the interfaces provided by the system or customize them. |

|   | Disadvantages                        | Time and effort needed to implement these services. | If it is necessary to customize user attributes or the type of authentication the code of the API should be manipulated. As the API is delivered as an Android library project and a web dynamic project, the code is always available. |
|   |                                      | The developer should know how to use Android facilities to implement these services. | If the code is manipulated to customize these services, updates of the SyncAndroid system will not work easily. |

Table 7-2. ToDo Lite-Authentication and Synchronization services. Hand coded vs SyncAndroid.

7.2.4. Performance
As claimed in requirement SY-NFR-05, one of the goals of the system is to provide an API to interact with the database with minimal impact on application performance. This section analyses how an Android developer can implement the functionalities of the application using the SyncAndroid system and manually coding all components.

Figure 7-3 presents the results of performance tests conducted in the SyncAndroid system. As data table of Figure 7-3 shows, the time needed to create documents in the database and to query documents from the database was measured for 10, 100 and 1000 records. The results show that the operations were slower using the API than performing these operation directly in the database, but, as it can be seen in the figure, the difference is not significant. Additionally, the results shows that when the same query is performed a second time, meaning that it is already in the cache of the system, it is even faster than performing the same query directly from the database. The data presented in this section leads to the conclusion that, as expected, the performance of the application is affected by the layer introduced by the system. Nonetheless the difference is not big, and it can even provide benefits when the same query is performed repeatedly, as it usually happens in the Android application when the user is navigating through interfaces.
7.2.5. Case Study Summary

As it can be seen in this section, the results of the case study are promising. It is difficult to provide an impartial opinion about the usability of the system because the developer is aware of all the facilities of the product. Nevertheless, the presented results provide strong indications that the use of the SyncAndroid system to develop Android applications with synchronization capability can save effort and time in the implementation process. Additionally, it makes it easier to implement this kind of applications, especially for novel Android developers since the system abstracts out complex concepts from the developer like NoSQL, network and Android services.
Additionally, the results of performance testing are promising because the difference of time to make database operations is not significant, and the use of the system cache can provide better performance when queries are performed repeatedly.

7.3. Summary

This chapter has presented the results of the evaluation of the final product achieved with this project. Section 7.1 describes the testing process performed on the system. First, the approach used to test the application is discussed. Then, most relevant test cases used in the process and their results are presented. Section 7.2 presents a demonstrator application developed using the SyncAndroid system. This application was developed in two different versions, one using the SyncAndroid System and a second version without using it. This exercise provided the opportunity to evaluate the usefulness of the system and the fulfilment of the non-functional requirements.
8.1. Project Review

The aim of this project was to implement a model-driven solution for generating efficient data-synchronisation layers for Android applications. To accomplish this aim, a thorough review of the domain area was conducted. Data synchronization is the key topic of the project, although a review of others topics should be performed in order to acquire the necessary knowledge and background information needed to face the challenges that the project raised. Furthermore, it is worth noticing that in order to counteract the lack of experience of the developer in most of the areas that comprise the domain of the project, a wide set of tools that provide similar or related services were analysed to obtain a solid background knowledge.

The research process described above lead to search for alternative ways to store information in mobile devices. Relational databases can be considered as the de facto standard for data storage in Android mobile devices but they raise several challenges for data synchronization since they were not conceived for distributed systems. This opened the possibility of using a NoSQL database for the project because of the support that they provide for information synchronization. This is emphasized here because of the novelty of this approach. No competitor products could be found that provides an API that abstracts out the logic of NoSQL databases from mobile developers. This raised the challenge of implementing a solution merging different tools and technologies that are usually used in different domains and the knowledge obtained during the Master Programme modules in Software Engineering such as Model Driven Development, Web Services, Software Testing, Mobile Application Development, Project Management, etc.

After the literature review process, the development process began with the requirements analysis. This analysis allowed to produce a set of stakeholder requirements which were elicited in order to produce the system requirements. Nonetheless, besides the use of a requirement engineering approach to define a complete set of requirements before starting the development process, additional system requirements were elicited as the project progressed in order to provide a better final product.

The design and implementation phases were conducted concurrently in different iterations using the RAD methodology. Nonetheless, defining the architecture at an early stage provided several advantages to implement the system, the initial design had to be revisited after each iteration in order to solve mistakes caused by the limited experience of the developer in the application of some of the technologies used during the implementation. This also provided the opportunity to not only aggregate new functionalities in each iteration, but also to improve the components that were already implemented in previous iterations.

During each iteration tests were conducted in each component using a Bottom-Up approach, allowing to detect most of the bugs of the system at early stages making it easier to detect and correct these bugs. At the final iteration the integration test was emphasized in order to ensure the correctness of the delivered product. During the test phase several difficulties were found to automate the integration test cases specified for the system because of the distributed nature of it. Test cases define procedures that should be performed in different platforms (Android application and web servers) co-ordinately. These difficulties leaded to give great importance to the test of each component and to define test cases that could be conducted manually in an efficient way.

Finally, a demonstrator application was implemented using the final product and a hand coded version of it was implemented in order to evaluate the usefulness of the final product. It was difficult to provide an impartial opinion about the usability of the system because the developer is aware of all the facilities of the product. Nevertheless, the obtained results provide strong indications that the use of the
SyncAndroid system to develop Android applications with synchronization capability can save effort and time in the implementation process, and it makes it easier to implement this kind of applications. This can be especially important for novel Android developers. Also should be noted that the affectation of the application performance is not significant.

8.2. Future Work

This section presents areas of possible further work in order to improve the system.

8.2.1. Queries

Couchbase Lite database uses the map-reduce approach to query data from the database. The system provides facilities to retrieve information from the database abstracting out the difficulties of the map-approach from the developer. Nevertheless, only basic functions are provided to filter data. Even though these functionalities can be enough to fulfil the requirements of a wide variety of mobile applications, it can be useful to provide support for complex queries where composed indexes and aggregation functions could be used.

8.2.2. Security

The exchange of user account between the mobile application and the server does not use a strong protocol in terms of security since it uses HTTP Basic Authentication and cookie-bases sessions. The implementation of HTTPS (Hypertext Transfer Protocol Secure) is an improvement that should be made to the system in order to provide a secure data exchange between the components of the system.

8.2.3. Support for Custom Indirect Authentication

Supporting alternative authentication system could provide several advantages for the developers and the end users of applications. For example, in a business environment where LDAP (Lightweight Directory Access Protocol) or Active Directory technologies are used to provide authentication services, the integration of them in mobile applications will lead to a simpler and centralized management of user accounts. Additionally, supporting indirect authentication using Google or Facebook accounts easier to use the synchronization services provided by the system.

8.2.4. Auto-fixing in Model Validation

The system validates the data model provided by the developer to build the data API and when critical errors that invalidate the model are found, the system interrupts the code generation process and displays messages to guide the developer to find and correct these errors. Nevertheless, the system does not make use of semi-automated fixing capabilities and critiques that the Epsilon framework provides. Auto-fixing provides the developer with the opportunity to fix errors in the model using the recommendations provided by the system without the necessity of re-editing the model. This functionality is necessary in order to improve the functionality of the tools.

8.3. Personal Growth

The project allowed me not only to consolidate the knowledge gained during the Software Engineering Master’s Programme, but it also gave me the opportunity to revaluate some technical concepts that I have been using for a long time in my professional life in a wrong way and it allowed me to gain confidence to face challenges in domains where I have limited knowledge or are totally new for me.

Model Driven Development is one of the areas that allowed to reformulate my understanding of software development. MDD allow me, as a developer, to design systems at a higher level of abstraction where the implementation details are not the concern, thus producing a software that is faster and with fewer errors than with traditional development approaches. The use of this approach also made me recognize the importance of the development of reusable codes and a good architecture design.
Another important conclusion is how important it is to generate a well-supported architecture design before starting the implementation process. In this project, where different technologies and tools were used, the definition of an initial architecture was a difficult and complex work. Nevertheless, this initial effort provided the base that guided the implementation, even when important changes had to be made in the system in later stages. Of course I still consider that the architecture of a system should be flexible enough to accommodate unexpected changes. Another important experience related to software architecture is the consideration of non-functional requirements at early stages of the development process. Before this project, functional requirements were always my main concern and requirements like performance, reusability, etc. were pushed to a second place. Now I understand that non-functional requirements are as important functional requirements, and they should always be considered at early stages because they will drive the implementation and structure of the functional requirements.

The analysis of different development methodologies and the use of RAD led me to two main conclusions. First, the selection of a methodology should not be limited to the use of approaches where the developer has experience and feels confident. The selection of a methodology should evidently consider the experience of the development team in the use of deployment methodologies, but the main criteria should be the characteristics of the project such as the size of the team, the complexity of the product and the resources of the project. But it is also important to notice that a methodology should not be a straitjacket for the development process, it should be used as a guide that should be adapted to the special characteristics of the project and of the developer team.

Finally, I should say that this project allowed me to gain confidence to face new projects. Before this experience, my work was always limited to the simple adaptation of well-known tools and technologies to specific domains. This project gave me the experience to apply new complex concepts and technologies in unknown areas for me. This is very important for me, because before this experience research was only a remote concept, but know I see it as something that is not only possible for me, but it is also something that I can enjoy.
BIBLIOGRAPHY


[34] Couchbase open source project, “Making the Shift from Relational to NoSQL.” Couchbase open source project.


[47] Ben Scofield, “NoSQL @ CodeMash 2010,” 06:25:21 UTC.


92

# TABLE OF APPENDIXES

<table>
<thead>
<tr>
<th>APPENDIX</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPENDIX A</td>
<td>B</td>
</tr>
<tr>
<td>GLOSSARY OF ABBREVIATIONS</td>
<td>B</td>
</tr>
<tr>
<td>APPENDIX A-1</td>
<td>B</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>C</td>
</tr>
<tr>
<td>SYNCANDBOX DESIGN</td>
<td>C</td>
</tr>
<tr>
<td>APPENDIX B-1</td>
<td>C</td>
</tr>
<tr>
<td>ToDo Detailed Class Diagram</td>
<td>C</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>D</td>
</tr>
<tr>
<td>SYNCANDBOX IMPLEMENTATION</td>
<td>D</td>
</tr>
<tr>
<td>APPENDIX C-1</td>
<td>D</td>
</tr>
<tr>
<td>Configuration of Code Generator Extension</td>
<td>D</td>
</tr>
<tr>
<td>APPENDIX C-2</td>
<td>D</td>
</tr>
<tr>
<td>APPENDIX C-3</td>
<td>E</td>
</tr>
<tr>
<td>Synchronization Configuration Function Template</td>
<td>E</td>
</tr>
<tr>
<td>APPENDIX C-4</td>
<td>F</td>
</tr>
<tr>
<td>Synchronization Configuration Function for Model TODO</td>
<td>F</td>
</tr>
<tr>
<td>APPENDIX C-5</td>
<td>H</td>
</tr>
<tr>
<td>Manifest Configuration Snippet for TODO Model</td>
<td>H</td>
</tr>
<tr>
<td>APPENDIX C-6</td>
<td>I</td>
</tr>
<tr>
<td>Authentication Configuration File Example for TODO Model</td>
<td>I</td>
</tr>
<tr>
<td>APPENDIX C-7</td>
<td>I</td>
</tr>
<tr>
<td>SyncAdapter Configuration File Example for TODO Model</td>
<td>I</td>
</tr>
<tr>
<td>APPENDIX D</td>
<td>J</td>
</tr>
<tr>
<td>TESTING APPENDIXES</td>
<td>J</td>
</tr>
<tr>
<td>APPENDIX D-1</td>
<td>J</td>
</tr>
<tr>
<td>Model with Errors Used for Validation Testing</td>
<td>J</td>
</tr>
<tr>
<td>APPENDIX D-2</td>
<td>K</td>
</tr>
<tr>
<td>Blog Model Used for Testing</td>
<td>K</td>
</tr>
<tr>
<td>APPENDIX D-3</td>
<td>L</td>
</tr>
<tr>
<td>APPENDIX D-4</td>
<td>L</td>
</tr>
<tr>
<td>ToDo Data Model Used for Testing and Evaluation Application</td>
<td>L</td>
</tr>
<tr>
<td>APPENDIX D-5</td>
<td>M</td>
</tr>
<tr>
<td>Authenticator.xml Generated from ToDo Data Model Used for Testing and Evaluation Application</td>
<td>M</td>
</tr>
<tr>
<td>APPENDIX D-6</td>
<td>M</td>
</tr>
<tr>
<td>sync_adapter.xml Generated from ToDo Data Model Used for Testing and Evaluation Application</td>
<td>M</td>
</tr>
<tr>
<td>APPENDIX D-7</td>
<td>M</td>
</tr>
<tr>
<td>Manifest-snipet Generated from ToDo Data Model Used for Testing and Evaluation Application</td>
<td>M</td>
</tr>
<tr>
<td>APPENDIX D-8</td>
<td>O</td>
</tr>
<tr>
<td>Manifest.xml File used with ToDo Data Model Used for Testing and Evaluation Application</td>
<td>O</td>
</tr>
<tr>
<td>APPENDIX D-9</td>
<td>Q</td>
</tr>
<tr>
<td>config.json File used with ToDo Data Model Used for Testing and Evaluation Application</td>
<td>Q</td>
</tr>
<tr>
<td>APPENDIX D-10</td>
<td>S</td>
</tr>
<tr>
<td>List Fragment</td>
<td>S</td>
</tr>
<tr>
<td>APPENDIX D-11</td>
<td>T</td>
</tr>
<tr>
<td>Main Activity and Task Fragment</td>
<td>T</td>
</tr>
</tbody>
</table>
### Appendix A
#### Glossary of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACID</td>
<td>Atomicity-Consistency-Isolation-Durability</td>
</tr>
<tr>
<td>ADT</td>
<td>Android Development Tools</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create Read Update and Delete</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Tools</td>
</tr>
<tr>
<td>Epsilon</td>
<td>Extensible Platform of Integrated Languages for mOdel maNagement</td>
</tr>
<tr>
<td>EVL</td>
<td>Epsilon Validation Language</td>
</tr>
<tr>
<td>FDD</td>
<td>Feature-Driven Development</td>
</tr>
<tr>
<td>FR</td>
<td>Functional Requirement</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext Transfer Protocol Secure</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Developing Environments</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>MDSD</td>
<td>Model Driven Software Development</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multiversion Concurrency Control</td>
</tr>
<tr>
<td>NFR</td>
<td>Non-Functional Requirement</td>
</tr>
<tr>
<td>NoSQL</td>
<td>No only SQL</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development</td>
</tr>
<tr>
<td>RUP</td>
<td>Rational Unified Process</td>
</tr>
<tr>
<td>SD</td>
<td>Secure Digital</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SH</td>
<td>Stakeholder</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SY</td>
<td>System</td>
</tr>
<tr>
<td>SyncML</td>
<td>Synchronization Mark-up Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>UUID</td>
<td>Universal Unique Identifier</td>
</tr>
</tbody>
</table>
Appendix C
SyncAndroid Implementation

Appendix C-1
Configuration of Code Generator Extension

```xml
<extension point="org.eclipse.ui.popupMenus">
  <objectContribution id="com.york.cs.swe.syncdoc.ui.emfatic" nameFilter="*.*" objectClass="org.eclipse.core.resources.IFile">
    <menu label="SyncAndroid" path="additions" id="com.york.cs.swe.syncdoc.menu1">
      <separator name="group1" />
    </menu>
    <action class="com.york.cs.swe.syncdoc.popup.actions.SyncDocObjectActionDelegate" enablesFor="1" icon="icons/generateall.png" id="com.york.cs.swe.syncdoc.generatecode" label="Generate Code" menubarPath="com.york.cs.swe.syncdoc.menu1/group1">
      </action>
      <visibility>
      <or>
      <objectState name="name" value="*.emf" />
      </objectState>
      <objectState name="name" value="*.ecore" />
      </or>
      </visibility>
    </objectContribution>
  </extension>
```
Appendix C-3

Synchronization Configuration Function Template

```java
[import "javautil.eol";]
[import ".ecoreutil.eol";]
[)
]clazzez=EClass.all.select(sc | sc.isDocument());
{
  "facebook": { "register": true },
  "databases": {
    "todos": {
      "server": "http://localhost:8091",
      "bucket": "sync_gateway",
      "users": {
        "GUEST": {"disabled": true}
      },
      "sync": `function(doc, oldDoc) {
        if (doc.type == "com.york.cs.services.authentication.Profile.class") {
          channel("com.york.cs.services.authentication.Profile.class");
          var user = doc._id.substring(doc._id.indexOf(":")+1);
          if (user !== doc.userId) {
            throw({forbidden : "profile user_id must match docid"})
          }
          requireUser(user);
          access(user, "com.york.cs.services.authentication.Profile.class"); // TODO this should use roles
        }
        [for (c in clazzez.select(sc|sc.isShareable())){%
          else if (doc.type == "[%=c.ePackage.name%].[%=c.name%]"){
            channel("%=c.name%");
            if (!doc.owner_id) {
              throw({forbidden : "[%=c.name%] must have an owner"})
            }[ONLY FOR OWNER*]
            [if(c.isCanEditAll()){%
              if (oldDoc) {
                var oldOwnerName = oldDoc.owner_id.substring(oldDoc.owner_id.indexOf(":")+1);
                requireUser(oldOwnerName)
              }]
            [%]}
            var ownerName = doc.owner_id.substring(doc.owner_id.indexOf(":")+1);
            access(ownerName, "%+doc._id");
            if (Array.isArray(doc.membersId)) {
              var memberNames = [];
              for (var i = doc.membersId.length - 1; i >= 0; i--) {
                memberNames.push(doc.membersId[i].substring(doc.membersId[i].indexOf(":")+1))
              }
              access(memberNames, "%+doc._id");
            }
          [%]}
        [%]}
        [for(c in clazzez.select(sc|sc.isPrivate())){%
          else if (doc.type == "[%=c.ePackage.name%].[%=c.name%]"){
```
```
Appendix C-4
Synchronization Configuration Function for Model TODO

TODO data model can be seen in Listing 6-2.

```json
{
  "facebook": { "register": true },
  "databases": {
    "todos": {
      "server": "http://localhost:8091",
      "bucket": "sync_gateway",
      "users": {
        "GUEST": { "disabled": true }
      },
      "sync": "",
    }
  }
}
function(doc, oldDoc) {
  if (doc.type == "com.york.cs.services.authentication.Profile.class") {
    channel("com.york.cs.services.authentication.Profile.class");
    var user = doc.id.substring(doc.id.indexOf(":")+1);
    if (user !== doc.userId) {
      throw({forbidden : "profile user_id must match docid"})
    }
    requireUser(user);
  }
```
```javascript
access(user, "com.york.cs.services.authentication.Profile.class"); // TODO
this should use roles
}
else if (doc.type == "com.york.cs.todolite2.document2.ListTasks"){
  channel(""+doc._id);
  if (!doc.owner_id) {
    throw({forbidden : "ListTasks must have an owner"})
  }
  if (oldDoc) {
    var oldOwnerName = oldDoc.owner_id.substring(oldDoc.owner_id.indexOf(":")+1);
    requireUser(oldOwnerName)
  }
  var ownerName = doc.owner_id.substring(doc.owner_id.indexOf(":")+1);
  access(ownerName, ""+doc._id);
  if (Array.isArray(doc.membersId)) {
    var memberNames = [];
    for (var i = doc.membersId.length - 1; i >= 0; i--) {
      memberNames.push(doc.membersId[i].substring(doc.membersId[i].indexOf(":" )+1))
    }
    access(memberNames, ""+doc._id);
  }
}
else if (doc.type == "com.york.cs.todolite2.document2.Task"){
  channel(""+doc._id);
  if (!doc.owner_id) {
    throw({forbidden : "Task must have an owner"})
  }
  if (oldDoc) {
    var oldOwnerName = oldDoc.owner_id.substring(oldDoc.owner_id.indexOf(":")+1);
    requireUser(oldOwnerName)
  }
  var ownerName = doc.owner_id.substring(doc.owner_id.indexOf(":")+1);
  access(ownerName, ""+doc._id);
  if (Array.isArray(doc.membersId)) {
    var memberNames = [];
    for (var i = doc.membersId.length - 1; i >= 0; i--) {
      memberNames.push(doc.membersId[i].substring(doc.membersId[i].indexOf(":" )+1))
    }
    access(memberNames, ""+doc._id);
  }
}
```

Appendix C-5
Manifest Configuration Snippet for TODO Model

<!-- add this configuration to your manifest file -->

<!-- permissions ==> before application configuration-->
<!-- network -->
<uses-permission android:name="android.permission.INTERNET" />

<!-- Authenticator -->
<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
<uses-permission android:name="android.permission.USE_CREDENTIALS" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />

<!-- Sync Adapter -->
<uses-permission android:name="android.permission.READ_SYNC_STATS" />
<uses-permission android:name="android.permission.READ_SYNC_SETTINGS" />
<uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS" />

<!-- permissions ==> inside application configuration-->

android:name="com.york.cs.todolite2.document2.Application"
android:allowBackup="true"
<activity
    android:name="com.york.cs.services.activities.LoginActivity"
    android:label="@string/loginLabel" />
<activity
    android:name="com.york.cs.services.activities.SignUpActivity"
    android:label="@string/sing_upLabel" />

<service
    android:name="com.york.cs.services.authentication.AuthenticatorService" >
    <intent-filter>
        <action android:name="android.accounts.AccountAuthenticator" />
    </intent-filter>
    <meta-data
        android:name="android.accounts.AccountAuthenticator"
        android:resource="@xml/authenticator" />
</service>

<!-- Sync adapter related service -->
<service
    android:name="com.york.cs.services.SyncAdapter.SyncAService"
    android:exported="true" >
    <intent-filter>
        <action android:name="android.content.SyncAdapter" />
    </intent-filter>
    <meta-data
        android:name="android.content.SyncAdapter"
        android:resource="@xml/sync_adapter" />
</service>
appendix c

authentication configuration file example for todo model

todo data model can be seen in listing 6-2.

<provider
    android:name="com.york.cs.services.SyncAdapter.ContentProviderApp"
    android:authorities="com.york.cs.todolite2.document2"
    android:exported="false"
    android:syncable="true" />

appendix c-7

syncadapter configuration file example for todo model

todo data model can be seen in listing 6-2.

<sync-adapter
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:contentAuthority="com.york.cs.todolite2.document2"
    android:accountType="com.york.cs.todolite2.document2"
    android:userVisible="true"
    android:allowParallelSyncs="false"
    android:isAlwaysSyncable="true"
    android:supportsUploading="true"/>
package com.york.cs.todolite2.test.blog;

class Blog {
  val Post[*] posts;
  val Member[*] _members; // ==> error name
  val Author[*] authors;
}

class Post {
  attr String title;
  attr String[*] tags1;
  attr int[*] ratings;
  val Comment[*] comments;
  ref Author author1;
  ref Author[*] authors2;
  val Stats stats;
  ref PostType[*] postTypeRef; //==> error reference
  attr PostType postType;

  enum PostType {
    Regular;
    Sticky;
  }

  class Stats {
    attr int pageloads;
    attr int visitors;
  }

  class Comment {
    attr String text;
    ref Author author;
    @image //==>error attachment in no document
    val Comment[*] replies;
    ref Member[*] liked;
    ref Member[*] disliked;
    attr Flag[*] flags;
  }

  enum Flag {
    Helpful;
    Offensive;
    Spam;
  }

  class Person {
    attr String name;
  }

Author extends Person {
}

Member extends Person {
}

class Blog {
    val Post[*] posts;
    val Member[*] members;
    val Author[*] authors;
}

class Post {
    attr String title;
    attr String[*] tags1;
    attr int[*] ratings;
    val Comment[*] comments;
    ref Author author1;
    ref Author[*] authors2;
    val Stats stats;
    attr PostType postType;
}

enum PostType {
    Regular;
    Sticky;
}

class Stats {
    attr int pageloads;
    attr int visitors;
}

class Comment {
    attr String text;
    ref Author author;
    val Comment[*] replies;
    ref Member[*] liked;
    ref Member[*] disliked;
    attr Flag[*] flags;
}

enum Flag {
    Helpful;
    Offensive;
    Spam;
}
class Person {
    @searchable
    attr String name;
}

class Author extends Person {
}

class Member extends Person {
}

Appendix D-3
Appendix D-4
ToDo data Model Used for Testing and Evaluation Application

package com.york.cs.todolite2.document2;

class TodoDB {
    val ListTasks[*] listTasks;
    val Task[*] tasks;
}

class ListTasks {
    attr String title;
    ref Task[*] tasks;
}

class Task {
    attr String title;
    attr boolean checked;
    @image
    attr String image;
    val Location[1] location;
}

class Location {
    attr String name;
    attr String addres;
}
Appendix D-5
authenticator.xml Generated from ToDo data Model Used for Testing and Evaluation

```xml
<?xml version="1.0" encoding="utf-8"?>
<account-authenticator
xmlns:android="http://schemas.android.com/apk/res/android"
android:accountType="com.york.cs.todolite2.document2"
android:icon="@drawable/ic_launcher"
android:label="@string/app_name"
android:smallIcon="@drawable/ic_launcher"/>
```

Appendix D-6
csync_adapter.xml Generated from ToDo data Model Used for Testing and Evaluation

```xml
<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
android:contentAuthority="com.york.cs.todolite2.document2"
android:accountType="com.york.cs.todolite2.document2"
android:userVisible="true"
android:allowParallelSyncs="false"
android:isAlwaysSyncable="true"
android:supportsUploading="true"/>
```

Appendix D-7
Manifest-snipet Generated from ToDo data Model Used for Testing and Evaluation

```
<!-- add this configuration to your manifest file -->
<!-- permissions ==> before application configuration-->
<!-- network -->
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
<uses-permission android:name="android.permission.USE_CREDENTIALS" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
<uses-permission android:name="android.permission.READ_SYNC_STATS" />
<uses-permission android:name="android.permission.READ_SYNC_SETTINGS" />
<uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS" />
```
<!-- permissions ==> inside application configuration-->

    android:name="com.york.cs.todolite2.document2.Application"
    android:allowBackup="true"

<activity
    android:name="com.york.cs.services.activities.LoginActivity"
    android:label="@string/loginLabel" />

<activity
    android:name="com.york.cs.services.activities.SignUpActivity"
    android:label="@string/sing_upLabel" />

<service
    android:name="com.york.cs.services.authentication.AuthenticatorService" >
    <intent-filter>
    <action
        android:name="android.accounts.AccountAuthenticator" />
    </intent-filter>

    <meta-data
        android:name="android.accounts.AccountAuthenticator"
        android:resource="@xml/authenticator" />
</service>

<!-- Sync adapter related service -->

<service
    android:name="com.york.cs.services.SyncAdapter.SyncAService"
    android:exported="true" >
    <intent-filter>
    <action
        android:name="android.content.SyncAdapter" />
    </intent-filter>

    <meta-data
        android:name="android.content.SyncAdapter"
        android:resource="@xml/sync_adapter" />
</service>

<provider
    android:name="com.york.cs.services.SyncAdapter.ContentProviderApp"
    android:authorities="com.york.cs.todolite2.document2"
    android:exported="false"
    android:syncable="true" >
</provider>
Appendix D-8
Manifest.xml file used with ToDo data Model Used for Testing and Evaluation Application

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
	package="com.york.cs.todolite2"
	android:versionCode="1"
	android:versionName="1.0">

<uses-sdk
	android:minSdkVersion="17"
	android:targetSdkVersion="19" />

<uses-feature android:name="android.hardware.camera" />

<!-- client -->
<uses-permission android:name="android.permission.USE_CREDENTIALS" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />

<!-- Authenticator -->
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />

<!-- Sync Adapter -->
<uses-permission android:name="android.permission.READ_SYNC_STATS" />
<uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS" />

<application
	android:name="com.york.cs.todolite2.document2.Application"
	android:allowBackup="true"
	android:icon="@drawable/ic_launcher"
	android:label="@string/app_name"
	android:theme="@style/AppTheme">

<activity
	android:name="com.york.cs.services.activities.LoginActivity"
	android:label="@string/loginLabel" />
<activity
	android:name="com.york.cs.services.activities.SignUpActivity"
	android:label="@string/sing_upLabel" />
<activity
	android:name="com.york.cs.todolite2.activities.MainActivity"
	android:label="@string/app_name" >

<intent-filter>
	<action
	android:name="android.intent.action.MAIN" />
<category
	android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
</manifest>
```
<activity
        android:name="com.york.cs.todolite2.activities.ShareActivity"
        android:label="@string/title_activity_share" />
</activity>

<service
        android:name="com.york.cs.services.authentication.AuthenticatorService" />
<intent-filter>
    <action
        android:name="android.accounts.AccountAuthenticator" />
</intent-filter>
<meta-data
        android:name="android.accounts.AccountAuthenticator"
        android:resource="@xml/authenticator" />
</service>

<!-- Sync adapter related service -->
<service
    android:name="com.york.cs.services.SyncAdapter.SyncAService"
    android:exported="true">
    <intent-filter>
        <action
            android:name="android.content.SyncAdapter" />
    </intent-filter>
    <meta-data
        android:name="android.content.SyncAdapter"
        android:resource="@xml/sync_adapter" />
</service>

<provider
    android:name="com.york.cs.services.SyncAdapter.ContentProviderApp"
    android:authorities="com.york.cs.todolite2.document2"
    android:exported="false"
    android:synctype="true">
    </provider>
</application>
</manifest>
config.json file used with ToDo data Model Used for Testing and Evaluation Application

```json
{
  "facebook": { "register": true },
  "databases": {
    "todos": {
      "server": "http://localhost:8091",
      "bucket": "sync_gateway",
      "users": {
        "GUEST": { "disabled": true }
      }
    },
    "sync": `function(doc, oldDoc) {
      if (doc.type == "com.york.cs.services.authentication.Profile.class") {
        var user = doc._id.substring(doc._id.indexOf("":")+1);
        if (user != doc.userId) {
          throw({forbidden: "profile user_id must match docid"})
        }
        requireUser(user);
        access(user, "com.york.cs.services.authentication.Profile.class");
        // TODO this should use roles
      }
      else if (doc.type == "com.york.cs.todolite2.test.blog.Post"){
        throw({forbidden: "list must have an owner"})
      }
      else if (doc.type == "com.york.cs.todolite2.test.blog.Author"){
        throw({forbidden: "list must have an owner"})
      }
      else if (doc.type == "com.york.cs.todolite2.test.blog.Member"){
        throw({forbidden: "list must have an owner"})
      }
    }
  }
}
```
if (oldDoc) {
    var oldOwnerName = oldDoc.owner_id.substring(oldDoc.owner_id.indexOf(":")+1);
    requireUser(oldOwnerName)
}

var ownerName = doc.owner_id.substring(doc.owner_id.indexOf(":")+1);
access(ownerName, "\"+doc._id);
Appendix D-11
Main Activity and Task Fragment