BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

OFFLINE MODE SUPPORT IN MOBILE APPLICATIONS
PODPORA OFFLINE REŽIMU V MOBILNÍCH APLIKACÍCH

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

MAREK MUSIL

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. RADEK BURGET, Ph.D.

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2020/2021

Bachelor's Thesis Specification

23907

Student:	**Musil Marek**
Programme:	Information Technology
Title:	**Offline Mode Support in Mobile Applications**
Category:	Information Systems

Assignment:

1. Get acquainted with the available tools and libraries for creating mobile applications using web technologies. Focus especially on the React Native platform.
2. Study existing libraries for synchronizing mobile applications with the ability to work offline, such as Offix.
3. After consulting with the supervisor, design a general solution enabling the creation of mobile applications with the possibility of working in offline mode and delayed synchronization, including the resolution of possible conflicts.
4. Implement the proposed solution using appropriate technologies. Also implement a suitable sample application demonstrating the use of the proposed solution.
5. Test the resulting application.
6. Evaluate the achieved results.

Recommended literature:

- Anderson, N. J.: Getting Started with NativeScript. Birmingham: Packt Publishing, 2016
- React Native: https://facebook.github.io/react-native/
- Offix: https://offix.dev/

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Burget Radek, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 12, 2021
Approval date:	May 19, 2021

Abstract

The goal of this thesis is to research different approaches to mobile application development with a focus on the use of web technologies. Next, research and comparison of technologies that can be used to achieve offline support in mobile applications is done. As a later stage of the work, a showcase mobile application is designed and implemented with the use of researched technologies.

Abstrakt

Jedním z cílů této práce je průzkum různých přístupů k vývoji mobilních aplikací se zaměřením na využití webových technologií. Následně jsou zkoumány a porovnány existující řešení pro podporu offline režimu s opožděnou synchronizací dat u mobilních aplikací. Závěr této práce tvoří návrh a implementace mobilní aplikace, která využije zkoumané technologie.

Keywords

mobile applications, offline mode, offline first, React-Native, react native, Android, data synchronization

Klíčová slova

mobilni aplikace, offline rezim, React-Native, react native, Android, synchronizace dat

Reference

MUSIL, Marek. *Offline Mode Support in Mobile Applications.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Burget, Ph.D.

Offline Mode Support in Mobile Applications

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Radek Burget, Ph.D. The supplementary information was provided by Gianluca Zuccarelli and Wojciech Trocki from Red Hat. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Marek Musil
May 19, 2021

Acknowledgements

I would like to thank Gianluca Zuccarelli and Wojciech Trocki for their technical mentorship and for answering my questions even in the late-night hours.

Contents

Chapter 1

Introduction

The goals of this thesis are to discuss different approaches to mobile application development with a focus on the use of web technologies and on options of how to handle data synchronization with offline mode support. After research into background topics, the end goal is to create an application with the tools that were researched, that would support offline mode and has conflict resolution implemented.

The initial motivation for the thesis was an open source Aerogear community project backed by Red Hat Inc. named Offix, and its need of a better quality example application that would showcase its capabilities using the React Native framework. The Aerogear community was formed around projects that were planned to be part of a Red Hat Inc. product called Red Hat Managed Integration, which was created partially as a successor to the Red Hat Mobile Application Platform. This initiative was later abandoned due to market changes and the work transformed into other projects and products.

The content of this thesis is divided into several chapters. In Chapter 2 can be found a brief introduction to mobile application development and how web technologies can be used. Chapter 3 describes the technologies used for data synchronization and the technologies by which the offline mode support is achieved. The following Chapter 4 contains the initial markups for the app and explains the actual implementation. Afterwards, Chapter 5 focuses on research into alternative technologies to the Offix library and in the last Chapter 6, a conclusion can be found.

Chapter 2

Mobile application development

Over the past few years, mobile applications became more prominent and generally, people tend to use mobile phones instead of a desktop PC for Internet use. Various sources indicate that more than a half of all Internet traffic is generated by mobile users [12]. The growth of the mobile market has led to creation of many frameworks that make the overall development process easier, for example by sharing one code base for both major mobile platforms (Google's Android and Apple's iOS). Such frameworks usually use web technologies like HTML, CSS, JavaScript, or any other of their derived programming languages, such as TypeScript, for building the applications, instead of a native programming language for each platform.

As hinted in the previous paragraph, several approaches to mobile application development exist. Such approaches are based on the type of the application that is being developed. It is common that mobile applications are classified into Native, Hybrid and Web-based. Each type of application is suitable for a different scenario and has its advantages and disadvantages.

2.1 Native

Native applications are written in programming languages specific to the targeted platforms. For Android, the native language was Java, but since Google's announcement at Google I/O 2019, Kotlin has become the primary language to use. For Apple devices, there are two options. The older Objective-C is being pushed more into the background by the more modern Swift.

Generally, native applications are suitable where performance plays a big role, for example mobile games. The user experience is usually better, since the used components fit into the UI style of the whole operating system and that makes it easier for users to get familiar with the application. Native applications, unlike the other types, are not limited by the capabilities of the frameworks, which provide functions that can be used on multiple devices, but can fully use all of the built-in functions of the device.

However, native applications come with a few disadvantages for development. The most significant is the fact that the code base can be used only for one platform and, if there's a need for the application to work on more than one platform, the whole code base has to be re-written in a different language. This comes with a higher cost in both time and money. Native programming languages require more knowledge, because there are not only differences in the used programming language, but also in the API of the functionality commonly available on mobile devices and platforms. A lot can be achieved with the use of external libraries, but then, a library used for one platform may be missing an equivalent library for the other platform.

2.2 Hybrid

Hybrid applications most commonly use a mix of native and web technologies. Generally, hybrid applications have a shorter development process, because of the use of web technologies, and they have the ability to reuse the same code base for multiple platforms. Also, they do not require that big of an expertise in the mobile development area. One set of technologies can be used for both iOS and Android and also one team of developers can create both mobile and web applications, resulting in cheaper development and shorter time to market. That is especially important, because of how quickly the market evolves and it is important to get the application to the market before the competition does. Hybrid apps can access native functions like camera, GPS, microphone to some extent, however this depends on the framework used and its capabilities.

Of course, the hybrid approach comes with disadvantages just like the native approach.

- The most significant is performance. Hybrid applications are generally slower as they have to render WebView with the content. WebView acts as a browser window inside the application.
- It is quite challenging to debug the application thanks to the extra layer created by the framework. For example, during my React Native development, I have spent hours on debugging, and when I finally gave up and wrote the code as a React web application, I instantly saw error messages that were actually saying something useful and were easy to understand.
- User experience, as previously mentioned in Subsection 2.1, is much better in case of native applications. For Hybrid applications it is not as easy to maintain the balance of user experience for both platforms at the same time.
- Framework developers need time to catch up and implement support for features introduced by mobile phone manufacturers the platform operating system itself. Which means that hybrid frameworks are always a little behind. For example, I was not able to find out how a hybrid application can use a multi-camera system which is very popular on the market today.

One of the most well-known hybrid frameworks is Adobe PhoneGap (which was recently discontinued [1]) and its open source fork, Apache Cordova. Apache Cordova allows a very easy way to create a multi-platform application using HTML, CSS and JavaScript. It provides a way to access native functionality through plugins. The whole Cordova ecosystem is quite extensive and includes many other tools and frameworks. One tool worth mentioning here is Ionic for its popularity. Ionic is a front-end SDK (Software Development Kit) designed for a better user experience which allows developers to use front-end frameworks such as Vue.js, Angular or React[1]. Even though Ionic is very popular with more than 44 thousands stars on Github[2] and more than 44 thousands Weekly Downloads of Ionic

[1]Ionic - https://ionicframework.com/

[2]GitHub repository of Ionic Framework - https://github.com/ionic-team/ionic-framework

CLI from npmjs.com[3], according to Google Trends, it is being more and more overshadowed by React Native and rising Flutter[4].

2.3 Web-based

Web-based applications are web sites written in HTML, CSS or JavaScript, accessed from the web browser. Their biggest advantage is the short development time, low cost and ease of updates, because they do not need to go through the publishing process to the platform specific app store. That means that users cannot choose whether they want to update the application or not, which solves the problem that users keep delaying the updates for some reasons, because of previous negative s with updating their applications on desktop computers [6].

However, the disadvantages are often quite significant: web applications are usually too simple due to lower investments in them and do not provide many features in the way that native or hybrid applications can, for example access to camera, NFC, biometrics like fingerprint reader, etc. And the user experience is not as good either. The disadvantages are partially solved by Progressive Web Apps (PWA).

Progressive Web Apps

Progressive Web Apps (PWA), are web applications that can be bundled and installed like native or hybrid applications, except they use some variant of a web browser to show their content. Most often, they are distributed directly from the website with one press of a button. This could be a little bit problematic though, because users might not trust the developer website to add anything to their device.

PWAs for Android can be distributed via Google Play Store as well, thanks to Trusted Web Activity(TWA). TWA displays a full screen Chrome browser wrapped in Android application without the full UI (User Interface) of the browser. More detail can be found in the Chrome Developers Documentation[5].

The important part of PWA is the service worker. A service worker is a separate script running in the background that enables some features which are not available for a classic web page, such as offline access and push notifications. Caching is used to reduce load times, provide offline access by storing responses to requests and data [6]. Since the applications are using the installed web browsers on the device for rendering the content with WebView or other alternatives like TWA, they can be very small.

The Twitter Lite application is a good example of this. In comparison to the full native version, its APK [7] file is significantly smaller (26.24MB vs 1.09MB). In the case of Instagram by Facebook, the difference is even more significant. The native application for Android

[3]Ionic CLI is command line interface for creation of Ionic applications - https://www.npmjs.com/package/@ionic/cli

[4]Google Trends statistics comparing popularity of React Native, Flutter and Ionic - https://trends.google.com/trends/explore?date=2019-01-01%202021-04-01&q=%2Fg%2F11h03gfxy9,%2Fg%2F11f03_rzbg,%2Fg%2F1q6l_n0n0

[5]Trusted Web Activity on Chrome Developers - https://developer.chrome.com/docs/android/trusted-web-activity/overview/

[6]Progressive web apps (PWAs) - https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

[7]APK is the Android Package Kit. It is the file format for software packages that Android-based systems use for app distribution and installation.

uses 93.24 MB on the device, while the PWA variant uses only 246kB and the overall usage for user data is also significantly lower, although the features are significantly reduced too. For example, it is not possible to send videos in messages in the PWA version.

Limitations here are quite similar to web applications. Hardware sensors cannot be used at all, Bluetooth or biometrics are not available. Also their ability to use more data storage is both a disadvantage and at the same time an advantage. Limiting the storage limits even its capabilities, for example, for offline usage.

2.4 React Native

React Native is an open source framework developed and maintained by Facebook since its initial release in 2015. It enables mobile developers to create native-like applications with the use of JavaScript and React components. React is a JavaScript framework for building User Interfaces with the use of encapsulated components. UI (User Interface) and its behaviour are described in JSX (XML-like extension for JavaScript). React Native is difficult to categorize properly as either native or hybrid, because just like any hybrid framework, one programming language is used to create applications for both Android and iOS and some parts of the code base could be shared, but in the resulting application, components are rendered with the native code and interact with the native APIs.

React Native requires Node.js[8] and npm[9], which could be replaced by yarn[10]. The basic concept of React Native is that the JavaScript engine renders native components and communicates with the Native API through React Native Bridge. Communication between the JavaScript engine and the native platform's API is done by batched asynchronous calls for performance purposes[2]. The whole concept is visualized in Figure 2.1.

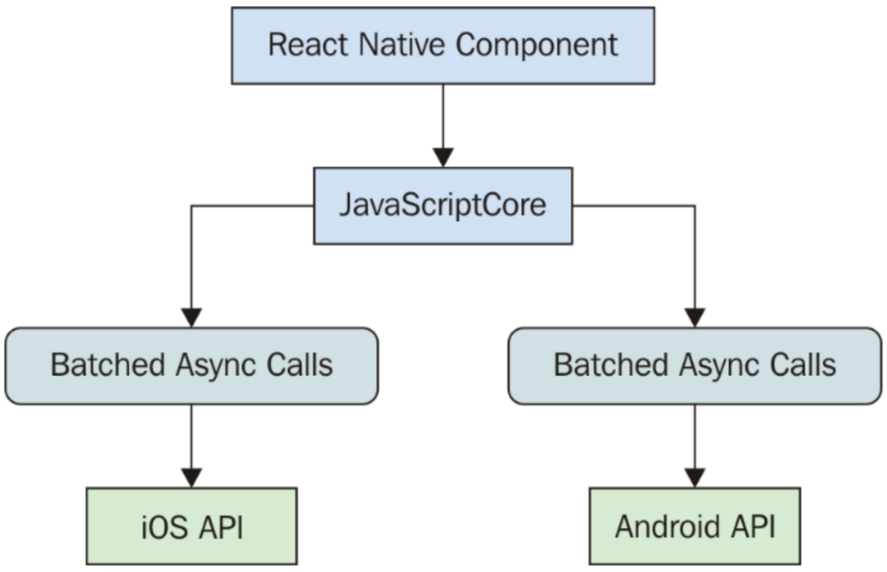


Figure 2.1: Visualization of React Native communication with native API. Figure from [2]

[8]Node.js - JavaScript runtime https://nodejs.org
[9]npm - package manager for Node.js https://www.npmjs.com
[10]Yarn - alternative package manager https://yarnpkg.com/

The application and its components are bundled and sent to the device with Metro[11]. Expo is a framework for building React and React Native applications and similarly to Metro it bundles the application, but also it handles more. With Expo, Xcode is no longer necessary for building iOS applications, so a developer does not need to own an Apple device to be able to build the application. It allows quick Over The Air updates for applications, so updates do not need to go through app store approvals. On top of that, Expo also simplifies the use of native libraries and linking them during a build. Expo is not a silver bullet though, because some native functionality are not available in applications created with it. For example Bluetooth, NFC or background location tracking.

2.5 Flutter

Flutter is another option for building cross-platform apps. It is an UI toolkit developed by Google Inc. and it was released in December 2018.

The programming language that is used is the strictly-typed Dart language. It is a relatively new language from Google, but is quite close to Java or C#, which makes it a little bit easier for developers to learn.

Originally, it was possible to create only Android and iOS applications, but with the recent Announcement of Flutter 2, the target platforms were hugely expanded with stable versions of Flutter Web, Desktop applications for macOS, Windows and Linux and also embedded systems in cars. Toyota has announced that they are planning to use Flutter 2 for their infotainment systems in cars, so it is safe to say that Flutter's popularity will only grow. Furthermore, some of Google's applications were migrated from native code to Flutter as well, for example Google Pay [3].

[11]Metro - JavaScript bundler for React Native https://facebook.github.io/metro/

Chapter 3

Data synchronization technologies

In this chapter, I will focus on technologies that are used for data synchronization in mobile applications, the existing solutions and a brief explanation how they work, to which I will also refer in Chapter 4 Implementation.

3.1 Basic concepts

The typical model for Data Synchronization involves having a local projection of the data that clients will consume independently of their network state. Client applications interact only with a local database that needs to be replicated to the server.

This approach brings a number of challenges. For example:

- Using a local database forces developers to consume all their data the same way even if the data itself is not going to be required when offline.
- As the data grows, the client side needs to perform more operations to replicate data that might not be even needed
- Even when online, clients can still be affected by the time required to replicate changes to the server and resolve possible conflicts that could be easily avoided when proper interaction with the server will be enabled.

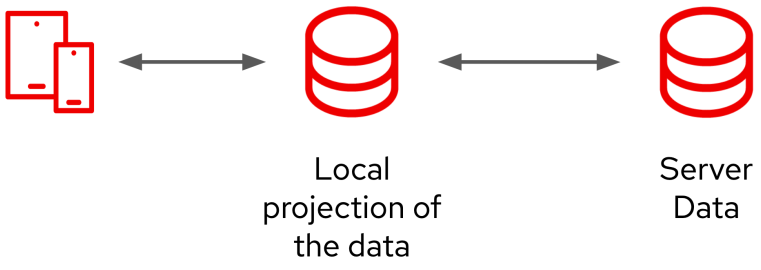


Figure 3.1: Basic model of Data Synchronization.

The modern approach to Data Synchronization gives developers ability to interact with the server and gives them access to the business logic. For example, instead of increasing the number of likes on Facebook post on the client and replicating it to the server (possibly causing conflicts) developers can execute business logic directly on the server to increment the counter.

What is important here is that application can work directly with the server by requesting both differences (diffs) as well as the entire dataset, while falling back to the local data in cases of network connectivity loss or of long latencies. This approach allows seamless interaction with the server while always keeping a local copy of the data.

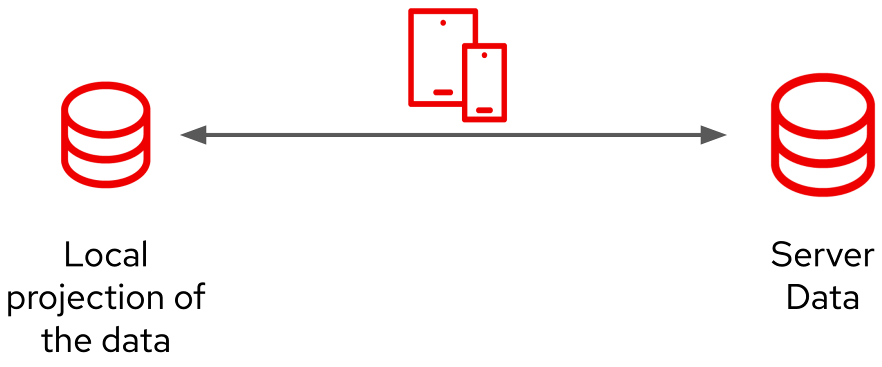


Figure 3.2: Modern model of Data Synchronization.

This model can be even extended to bring live updates straight into the application and store them in the local store. Developers can still request data directly from the server but they will also benefit from the separate client-side store.

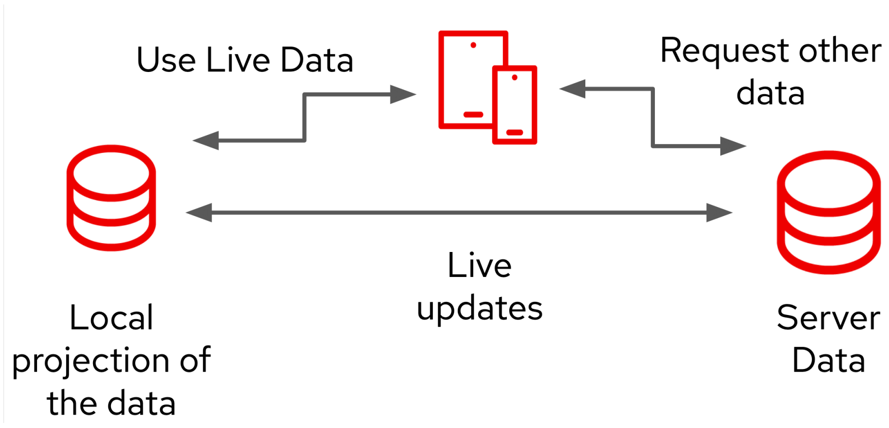


Figure 3.3: Model of Data Synchronization with realtime updates.

Basic functionalities of data synchronization solutions should include basic CRUD[1] operations, caching of changes while offline and execution of cached operations once the internet access becomes available, along with the ability to resolve potential conflicts which could occur once the device gets back online.

[1]CRUD operations - Create, Read, Update, Delete

3.2 Database systems

This section will be dedicated to the database systems which could be used with mobile applications and the differences between them. First, it is important to understand the basics of SQL and NoSQL databases and their main differences.

3.2.1 SQL database

SQL databases (or relational databases) are based on the relational model of data, which means that data are stored as relations. From the mathematical definition, we know that a relation is a set of n-tuples of values. In practice, it means that data are represented by tables with rows and columns, where rows represent tuples, and columns represent attributes. Attributes should contain only atomic values. The values are not complex values, but are simple ones and cannot be split any further.

Multiple tables are usually linked together by data common to both tables. Such data are known as candidate keys. The value of the attribute must be unique and cannot be reduced to a less complex form. As an example, we can imagine a table `Bank Account` and a table `Person` with a candidate key `National Identifier Number`. This candidate key is used as a foreign key in the `Bank Account` table, and uniquely links the bank account to its owner therefore together forming meaningful data.

The benefits of relational databases are mainly their maturity, their reduced redundancy of data and their simplicity of disaster recovery. They are also transactional, so at any given moment the data are consistent. In case some operation fails to complete, the transaction is canceled and the state of the system is as it was before the start of the transaction. Any reader of the data will never see the in-progress transaction, only the consistent previous or new states. This data consistency is of critical importance in some industries, for example in finance. There are a few disadvantages though. Relational databases could be a little slower with searching through huge amounts of data, and horizontal scaling could be complicated. Horizontal scaling is the ability to add more machines to the processing pool and therefore enable parallel execution.[2]

Typical SQL database systems are Oracle Database, MySQL and ProstgreSQL.

Structured Query Language

Structured Query Language (SQL) is the language used to interact with the relational database systems. The programmer forms queries, by which the user asks the server what data matches a query or defines an action to be performed. As a response, the user gets a table with requested data or information about the performed action. The queries and actions can be quite complicated and involve multiple tables or parts of tables, when using key attributes.

3.2.2 NoSQL database

NoSQL (not only SQL) databases provide a flexible data schema in order to be more agile, and generally do not use SQL for queries [5]. There are several approaches to how the data are stored in a NoSQL database. The most common according to the official MongoDB website [8] are the following:

[2]Horizontal scaling - https://www.section.io/blog/scaling-horizontally-vs-vertically/

- Document-based databases store data in a similar format to JSON (JavaScript Object Notation) object, which is a pair of keys and values that could have multiple types, such as numbers, strings, arrays, boolean or other objects. The most popular Document-based database is MongoDB[3].
- Key-value based databases use a simple approach to store data, each item is stored as a key and value pair, which could contain only primitive data types like numbers, booleans, strings, and arrays. The most popular key-value database is Redis[4].
- Wide Column-based databases store data like relational databases in tables with rows and columns, with the biggest difference being that columns are dynamic. That means that each row does not have to have the same columns. The most popular is Apache Cassandra[5].
- Graph databases store data as graphs with nodes and edges. Nodes store specific information and edges represent relationships between the nodes. According to the DB-Engines statistics, Neo4j[6] is the most popular Graph database and Microsoft Azure Cosmos DB is right in second place [11].

A great advantage of NoSQL databases over relational databases is that they do support horizontal scaling. MongoDB could serve as a good example, because there are two approaches that will be described in the next two paragraphs based on information from the official website [7].

Sharding

Sharding is basically splitting the data and distributing them over multiple nodes so each node stores only a part of the data. Sharding is very useful when we store big amounts of data, as each operation could be performed on a different node, therefore sharing load across the nodes. Such splitting could over time cause uneven distribution, but that is automatically solved by a shard balancer.

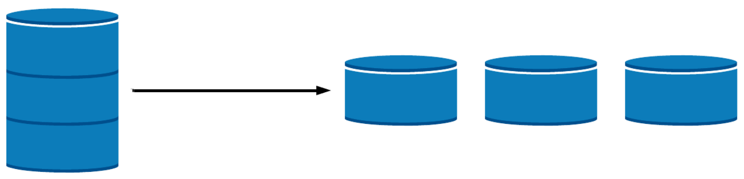


Figure 3.4: Depiction of sharding [7].

Replica sets

Replica sets, on the other hand, are only mirrors of the original dataset on multiple nodes. This is very useful in case one of the nodes fails, so data are backed up at any given moment. Replication also contributes to faster read times, as the data can be read from multiple machines at the same time, once again sharing load across the nodes.

[3]MongoDB - https://www.mongodb.com
[4]Redis - https://redis.io/
[5]Apache Cassandra - https://cassandra.apache.org/
[6]Neo4j - https://neo4j.com/

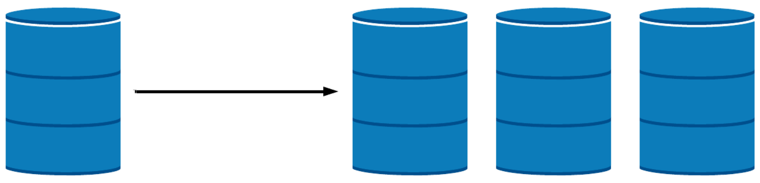


Figure 3.5: Depiction of replication [7].

3.2.3 Which type to choose

Since NoSQL databases are a better choice in cases when a more flexible data schema is necessary, they are a better choice for mobile applications. We do not need relations between the data for the resulting application and for simplicity, it will be much easier to use MongoDB and its JSON-like document approach to data storage.

3.3 GraphQL

GraphQL is a query language for APIs, creating a simplified way to request data from an endpoint. In case of a classic REST API, there are multiple endpoints, with specific purposes. GraphQL takes a different approach, that merges all the endpoints into a single one and the specific data that the client application needs is specified in the request. This makes it easier in cases where the application needs to get data from multiple endpoints at once.

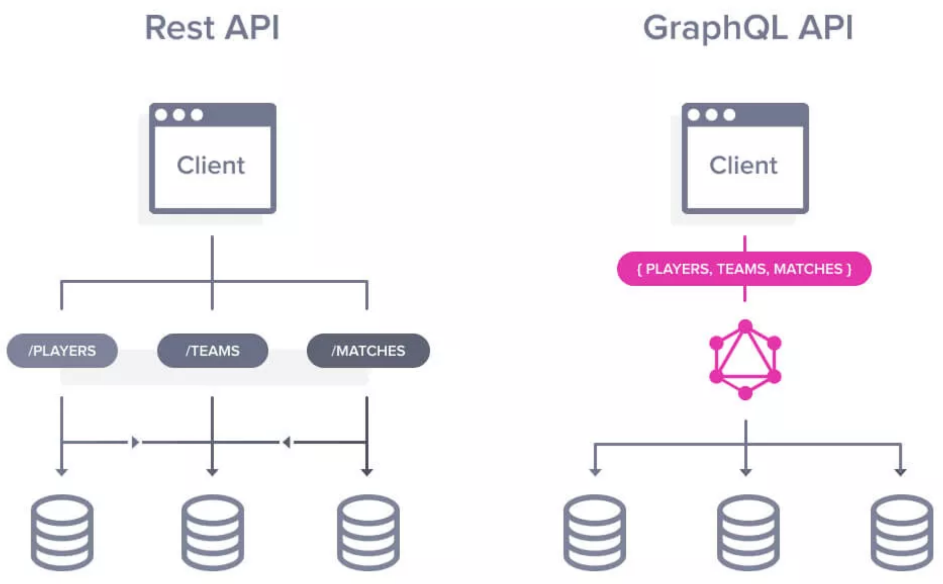


Figure 3.6: Comparison of REST API and GraphQL API [9].

Figure 3.6 depicts the difference in an example where we have three different types of data. For a REST API, three endpoints exist for each data type, while in case of GraphQL, there is only one endpoint which accepts specification of the data type in the request.

It is important to understand what GraphQL is, as it is essential technology used by many solutions for data synchronization.

3.3.1 Data schema

The basic building blocks in a GraphQL service are types and field on those types and functions for each field on those types. Fields are strictly typed, so each field needs to have specified data type of its content. The fields could be of various types, the classic scalar types such as Int, Float, String, Boolean or ID, Enumeration types, lists and other objects. It is even possible to use the abstract type *Interface* to use inheritance from another object.

3.3.2 Queries

GraphQL communicates with the client application with JSON or JSON-like objects. The query is quite simple. The examples are from the official GraphQL website [13].

```
{
  hero {
    name
  }
}
```

Listing 3.1: Query example.

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

Listing 3.2: Response example.

From Listing 3.1 we can see that the query reminds us of a JSON object. `hero` and `name` are referenced as fields and `hero` is also an object. With fields, we specify what data from the object we want to receive. If we add an `id` field to the object, we also get `id` in addition to the `name` of the object. The response from the API is a typical JSON object structured in the same format as the query. This way, the format of a response is always a predictable format, because it was already specified.

3.3.3 Mutations

A mutation is a method of writing data into the database. All write operations should be implemented as mutations, even though a query can write data as well. Listing 3.3 is an example mutation, which creates a review object in the database, and at the same time, returns the nested fields `starts` and `commentary` that were previously stored.

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

Listing 3.3: Mutation example [13].

3.3.4 Subscriptions

A subscription is a third operation type of GraphQL. It is fairly similar to the query, but unlike the query, subscription maintains the connection open for future updates and keeps fetching the data. This is often realized with the use of web sockets. Subscriptions are especially helpful when real time updates are needed in the application. A great example would be a chat application, where instant updates are essential. In cases where this is not as important, periodical polling and re-execution of the queries on a specific user action is better approach, since it does not generate as much network traffic.

3.3.5 Apollo GraphQL

Apollo GraphQL is a specific implementation of GraphQL providing both server and client tools for the GraphQL API. Apollo server can be used as a stand-alone GraphQL server or as an add-on to existing Node.js middleware[7]. Apollo client is available as a native Android or as an iOS library as well as a JavaScript library. It provides tools for fetching data, caching, and pagination. The JavaScript library is designed mainly to be used with React, but support for additional frameworks is achieved with 3rd party libraries, for example, Angular or Vue.js. Optimistic updates are supported as well.

Both Client and Server support File Uploads, although with Node 14 it gets a little more complicated and additional setup is required.

3.4 Graphback

Graphback is a tool created by the Aerogear community, which aims to significantly shorten the overall development time of GraphQL applications by auto-generation of the CRUD API and the use of back-end templates. It supports three different databases for storing the data: PostgreSQL for the relational databases and document-based NoSQL database MongoDB. The third one is SQLite, which is not a full-fledged database engine suitable for a bigger applications[8] and is not recommended for a production use by Graphback authors[9].

Graphback provides server side conflict resolution with a few build-in resolution strategies.

- `ClientSideWins` is a strategy where conflict will be resolved to whatever the client sent. If the object in conflict was deleted, it is restored to the state on the client. If

[7]Middleware is type of software that provides additional services and capabilities to applications that are not provided by the operating system. Middleware can be described as a glue between the applications. https://www.redhat.com/en/topics/middleware/what-is-middleware

[8]SQLite - more information can be found at https://www.sqlite.org/whentouse.html

[9]Using SQLite with Graphback - https://graphback.dev/docs/databases/overview

the conflict arises with a delete mutation, the object is deleted no matter its state on the server.

- `ServerSideWins` ensures that in the event of a update conflict, the client's update will never overwrite any field that has changed since the client last fetched it. If the object has been deleted in the database, the client will be notified of it by way of a `ConflictError`. For delete conflicts, the client is informed of the conflict via a `ConflictError`.

- `ThrowOnConflict` is strategy that throws `ConflictError` every time a client tries to change a field on an object that has been changed on the server while the client was offline.

- A custom `ConflictResolutionStrategy` can be implemented as well. As an example, we can use the `ClientSideWins` strategy from the Graphback library itself, as in Listing 3.4.

```
export const ClientSideWins: ConflictResolutionStrategy = {
  resolveUpdate(conflict: ConflictMetadata): any {
    const {serverData, clientDiff }= conflict

    const resolved = Object.assign(serverData, clientDiff);

    if (serverData[DataSyncFieldNames.deleted] === true) {
      resolved[DataSyncFieldNames.deleted] = false;
    }

    return resolved;
  },
  resolveDelete(conflict: ConflictMetadata): any {
    const {serverData, clientData }= conflict;

    if (serverData[DataSyncFieldNames.deleted] === true) {
      throw new ConflictError(conflict);
    }

    const resolved = Object.assign({}, serverData, {[
       DataSyncFieldNames.deleted]: true });

    return resolved
  }
}
```

Listing 3.4: Implementation of conflict resolution strategy.

3.4.1 Data model

As an input for Graphback we can use a GraphQL data model, represented as GraphQL types, from which the GraphQL schema is generated, including the GraphQL CRUD API(mutations and queries 3.3.2), resolvers, and client queries. To generate CRUD API for a GraphQL type, it only needs to be properly annotated with `"""@model"""` showcased by Listing 3.5, line 2. It is also possible to specify which CRUD API Graphback should create for each type. Basic relationships like one-to-many, one-to-one and many-to-many between GraphQL types are supported and to create them, field annotations are used.

```
1 """
2 @model
3 """
4 type Note {
5   id: ID!
6   text: String
7   description: String
8 }
```

Listing 3.5: Model example with annotation from Graphback website.

A more detailed description of a Data model with a more complex example can be found as Listing 4.1 in Section 4.1.1.

3.5 Offix

Offix was originally created as an extension to Apollo GraphQL's client library to provide offline support for applications. This approach was later abandoned and new approach was adopted. The following sub-chapters are dedicated to these two approaches.

3.5.1 Offix Client

The original Offix, Offix Client, supports all GraphQL operations while offline. Queries are targeted to the local cache of the data on the device, and mutations are stored in a queue. The default behaviour is that all queries are considered as offline queries, and when they are performed, the results are stored in the cache. To make sure mutations are performed on the latest version of the data, before each mutation the data are re-fetched first with either a cached query or a live query to the server. It is also possible to specify multiple queries that should be performed before a mutation. This is accomplished with the cache helpers and the `offix-cache` package.

```
import {CacheOperation }from ’offix-cache’;
import {findNotes }from ’../graphql/gql’;

const options = {
  updateQuery: findNotes,
  returnType: ’Note’,
};

export const add = {
  ...options,
  mutationName: ’createNote’,
  operationType: CacheOperation.ADD,
};

```

```
export const edit = {
  ...options,
  mutationName: 'updateNote',
  operationType: CacheOperation.REFRESH,
};

export const remove = {
  ...options,
  mutationName: 'deleteNote',
  operationType: CacheOperation.DELETE,
};
```

Listing 3.6: Mutation helper example.

Listing 3.6 is an example of the mutation helper from the implementation of the Note-taking application.

The Offix-cache package provides also subscription helpers for real time updates from the cached data. The implementation of the helpers is quite similar to mutation helpers. This means that a subscription is specified and also a query for updating the cached data. More concrete examples can be found in Section 4 Implementation.

Since mutations are not necessarily performed immediately, for a better user experience, the UI (User Interface) relies on Optimistic Responses. An Optimistic Response is a way to update the UI to reflect the performed changes even before the change was processed by the server. This approach is optimistic, because it expects that the operation will be performed successfully.

Caching and offline changes bring potential problems in terms of conflicting changes to the data, as it is quite common that the data spread over multiple devices will eventually become desynchronized and conflicting changes will start appearing. Offix provides tools for both client- and server-side conflict resolution. To properly detect and resolve potential conflicts, it is essential for the data to have additional information like version, timestamps of creation or last update. Developers can create their own resolution strategies that will be applied once the conflict is detected. By default, if not specified otherwise, the server will just apply client changes on top of its own data.

The Offix-Client approach comes with some disadvantages that I ran into even during my development. In case of more complex GraphQL schema with complex types, queries and relationships, it is necessary to create custom cache update functions, which could be highly difficult and requires a very good knowledge of Apollo GraphQL. This lead to a re-evaluation of the whole project and a separate DataStore was created to solve some of the main problems.

3.5.2 Offix DataStore

One of the main changes in Offix DataStore is that a local database is used for storing all offline data that offix-client previously stored as a cache. As for the specific database technology, for web browsers IndexedDB or WebSQL are supported and for mobile applications, SQLite is used. The database is based on the provided GraphQL model from the very beginning as the main data storage.

The GraphQLCRUD[10] specification is also supported, so the whole GraphQL schema with all queries and mutations can be generated without the need to create custom cache helpers, and so could be used out of the box with the Graphback server.

Offix DataStore is still under development, and is currently on a Beta version 0.5.0.

[10]GraphQLCRUD specification - https://graphqlcrud.org/docs/next/gettingstarted

Chapter 4

Implementation

For a showcase application, I have decided to create a simple Note-taking application that would support simple text- and list-type notes. Initial work was done with a combination of offix-client for the mobile, and a Graphback back-end.

4.1 Back-end

The back-end for the mobile application has been generated using Graphback from its template, containing MongoDB, DataSync configuration and Apollo GraphQL Server[1].

The generation process is depicted by Figure 4.1. The process is started with `yarn generate`, which only points to specified Graphback command. Graphback then takes the data model file stored in `model/datamodel.graphql` and generates the complete GraphQL schema file `schema.graphql` with a definition of all necessary scalars, input types, mutations, queries, type objects and subscriptions. Then with the use of a Codegen[2] TypeScript file `generated-types.tsx` with the schema is created in a representation that could be used with the server.

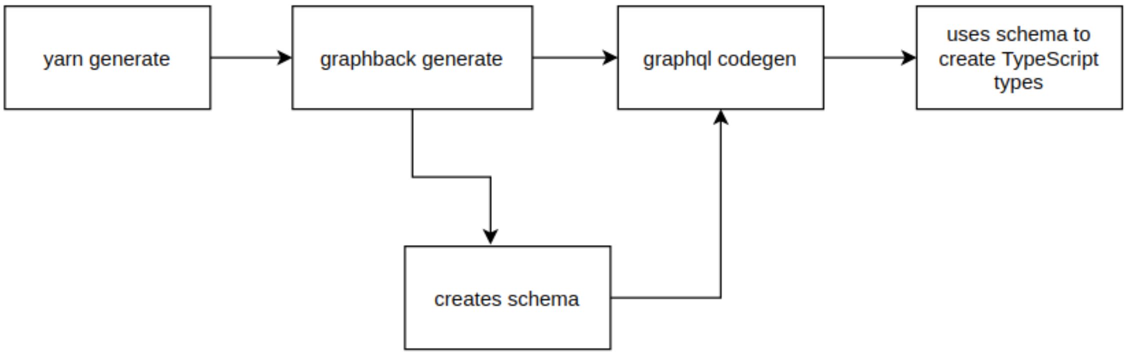


Figure 4.1: Code generation diagram provided with the template.

[1]Graphback back-end template https://github.com/aerogear/graphback/tree/master/templates/ts-apollo-mongodb-datasync-backend

[2]GraphQL Codegen is code generator for GraphQL schema https://www.graphql-code-generator.com/

4.1.1 Application Data model

The Data model consists of two main models. Type `Note` and type `Item`.

```
"""
@model
@versioned
@datasync(
  ttl: 5184000
)
"""
type Note {
  _id: GraphbackObjectID!
  title: String
  type: NoteTypes!
  version: Int!
  completed: Boolean

  text: String

  """
  @oneToMany(field: 'item')
  """
  list: [Item]
}
```

Listing 4.1: Note type

Listing 4.1 is a data model for Note, it is annotated with `@model` for auto-generation of CRUD API, `@versioned`, which adds fields with timestamps of creation and last update. This Note type is constructed as a universal type for both List and Text type notes. The text type will use text field for its content and List type note will use an array of Items which are linked to the note with the relationship type one-to-many. The field `type` is of NoteType data type, which is an enumeration with the set of values `"Text"` and `"List"`.

```
"""
@model
@datasync(
  ttl: 5184000
)
"""
type Item {
  _id: GraphbackObjectID!
  position: Int
  completed: Boolean
  text: String
}
```

Listing 4.2: Item type

From the models 4.1 and 4.2, it would appear, that in the actual database, Note would contain an array `list` with IDs of their nested Items. In reality though, Item records contain IDs of its parent Note. This I found very confusing, since query a returns a field `list` with an array of nested items. Listing 4.3 shows the structure of a List note that is returned by a query.

```
{
  "_id": "6090582de4a05a3ef090e1a7",
  "title": "List2",
  "type": "List",
  "version": 1,
  "completed": false,
  "createdAt": 1620072493688,
  "updatedAt": 1620072493688,
  "text": null,
  "list": [{
    "_id": "60905ecfe4a05a3ef090e1a8",
    "text": "item 2-1",
    "position": 1
    },
    {
    "_id": "60905ed9e4a05a3ef090e1a9",
    "text": "item 2-2",
    "position": 2
    },
    {
    "_id": "609b9ffe4a4d3b282a6af2bb",
    "text": "item 2-3",
    "position": 2
  }]
}
```

Listing 4.3: Item type

4.2 Mobile application

As noted above, the showcase mobile application is a Note-taking application that would support two note types, Text Note and List type. As a starting point for the application, I used the official example in the Offix Client library[3] for the necessary configuration to learn how to use it in the first place. The rest was built on top of it with React-Native components and the `react-native-paper` package[4], which provides some basic Material Design components.

4.2.1 User Interface

The User Interface of the application is inspired by the Material Design system, which is recommended for Android applications. React-Native can render the native components, but some more advanced components are distributed with external packages. After some

[3]Official Offix Client Example - https://github.com/aerogear/offix/tree/master/examples/react-native

[4]React-native-paper package - https://callstack.github.io/react-native-paper/

initial research, I discovered `React-native-paper`, which provides many components in Material design.

In the beginning, a simple prototype of the application was created with online tool FluidUI[5], which was a fast and easy way of prototyping the UI design.

Figure 4.2 is the prototype of the main page, which lists all notes as a list. This was changed later on during the implementation to accommodate more notes in a single view. The result is shown in Figure 4.3. The status bar was changed to a darker shade of orange and the button for adding new notes was changed to secondary color, in order to reflect more the color system of Material Design and elevate the button visually.

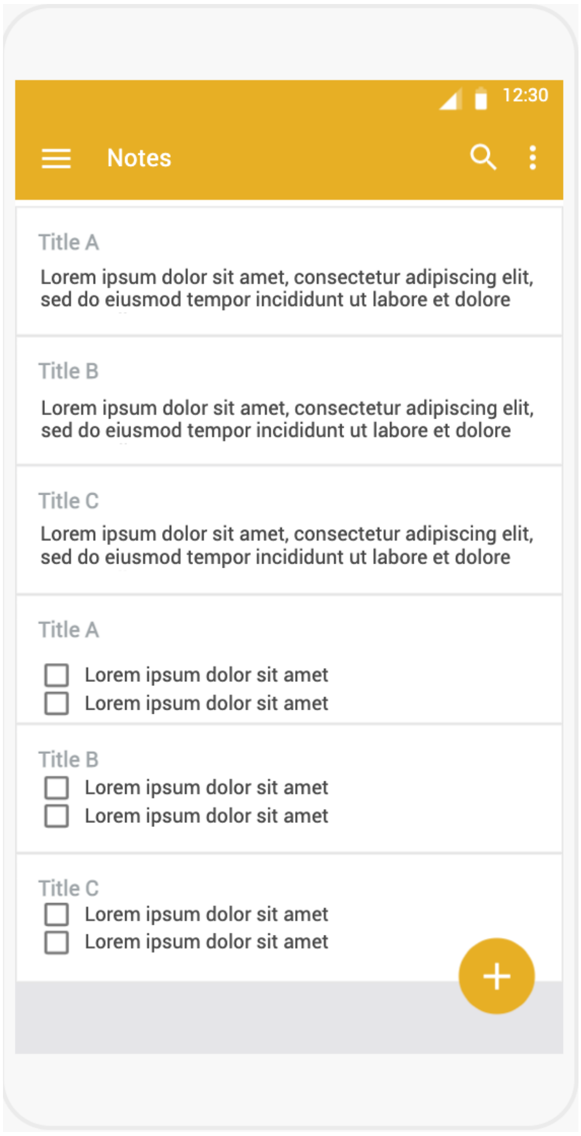
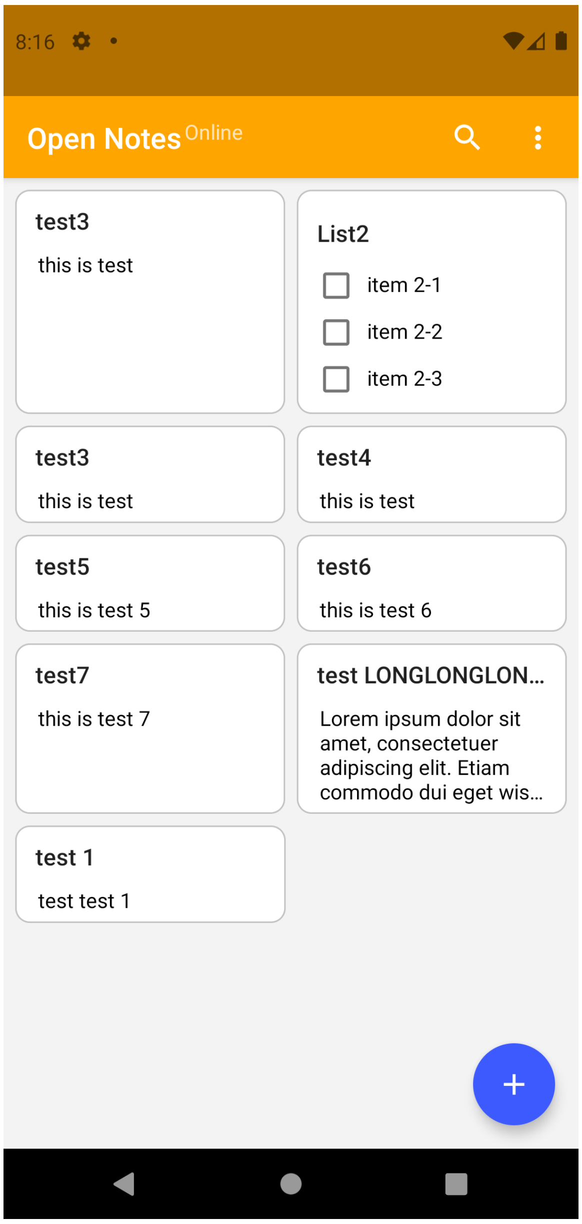

12:30
Notes
Title A
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
Title B
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
Title C
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
Title A
Lorem ipsum dolor sit amet
Lorem ipsum dolor sit amet
Title B
Lorem ipsum dolor sit amet
Lorem ipsum dolor sit amet
Title C
Lorem ipsum dolor sit amet
Lorem ipsum dolor sit amet

Figure 4.2: Prototype for the example application.

[5]FluidUI is online prototyping tool which provides an extensive library with UI components in the design for multiple platforms, and it uses mainly Material Design and iOS. More information can be found at https://www.fluidui.com/.



test3
this is test

List2
- [ ] item 2-1
- [ ] item 2-2
- [ ] item 2-3

test3
this is test

test4
this is test

test5
this is test 5

test6
this is test 6

test7
this is test 7

test LONGLONGLON...
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam commodo dui eget wis...

test 1
test test 1

Figure 4.3: Resulting UI for the example application.

The note creation screen is shown in Figure 4.4. As a default, Text Note type is selected and type can be changed with a press of the button with the title `CHANGE NOTE TYPE`.

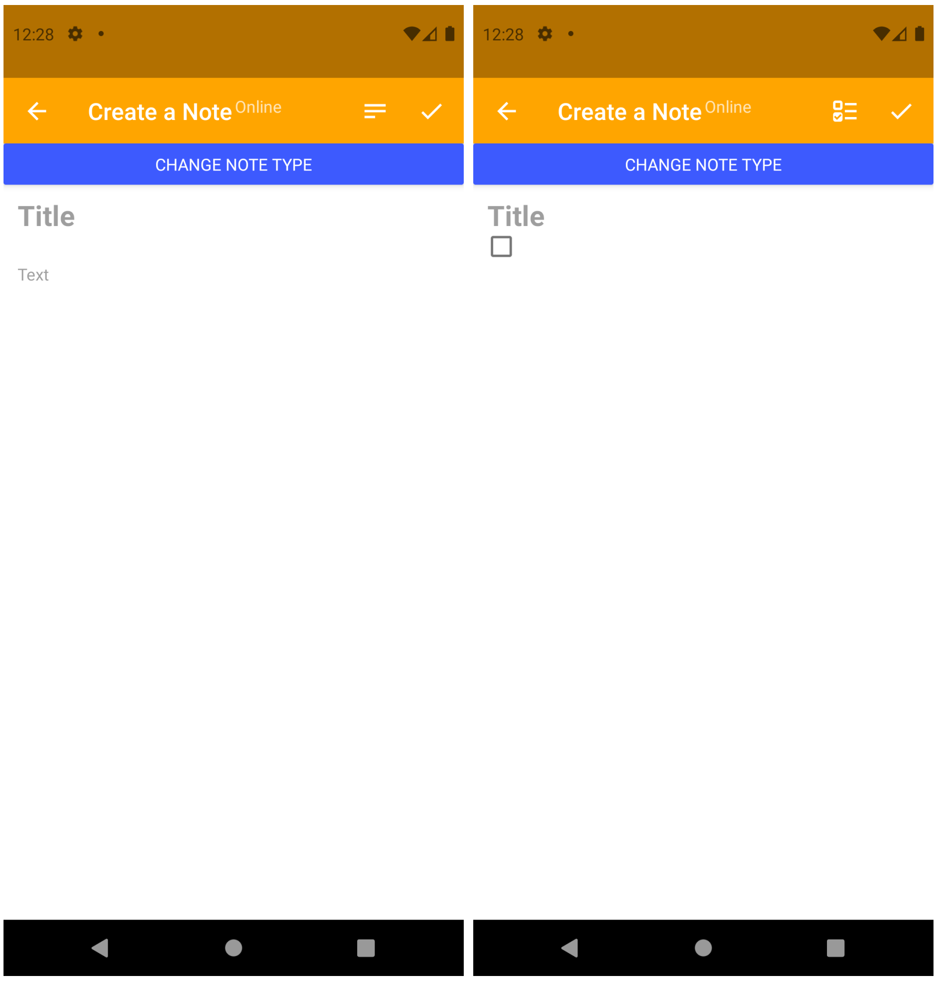


Figure 4.4: Modal window for creation of Note.

4.2.2 End state of the application

During the implementation, I have encountered a problem that I needed to discuss with my technical mentor at Red Hat. Our conversations lead us to the conclusion that, technically, my problem with caching of mutations is unsolvable with the current state of the Offix Client library, which was at the time being deprecated in favour of a new approach in development, which is described in Section 3.5.2. The problem was in the cache package itself, that it works for only very specific GraphQL schema, and that more complicated ones are not handled properly.

After further discussion on how to approach this we have decided that focusing on the Data Synchronization technologies as a research and comparison would be the most appropriate approach given the circumstances and state of the Offix library.

Chapter 5

Research into alternative libraries to Aerogear and Offix

There are several other existing solutions on the market for data synchronization for mobile applications, with both open source code and closed source code.

- Google Cloud Firestore is a cloud-hosted NoSQL database, which officially supports both Android and iOS mobile platforms, and also web applications. The document-oriented data model is used by Cloud Firestore and it also provides offline support for the main platforms. It provides integration with other Firebase products, such as Authentication or Cloud Functions [4]. It is also possible to use the Cloud Firestore for free with some limitations.[1] There is also a React-Native library, which offers the offline mode support as well. Nevertheless, the library is not officially supported by Firebase and does not work with Expo[2].

- RxDB is an alternative to Offix as it takes the GraphQL approach as well and provides tools for a real-time synchronization with an offline first approach. RxDB supports all major web browsers, Node.js, Electron[3], React-Native, Cordova and frameworks like Angular, Vue.js, React and Ionic. A very interesting feature is Multi-Window or Multi-tab synchronization which works completely offline. This is achieved by broadcasting state changes to all clients that are connected to the same storage engine. For the storage engines, RxDB provides multiple adapters that define where the data are stored. Both persistent and non-persistent options are available. A non-persistent option that can be used is JavaScript runtime memory, and there are multiple persistent options for different platforms. Just like Offix DataStore, IndexedDB or WebSQL are supported for web applications and for other platforms there are specific implementations of SQLite or React-Native's AsyncStorage [10].

- PouchDB is an in-browser database with the ability to save data locally to allow users to use the application even when they're offline. It also provides tools for synchronization between clients. PouchDB works with both web browsers and in Node.js and can be used as a direct interface to CouchDB-compatible servers. The API works the same way in every environment, so developers can spend less time

[1]Cloud Firestore pricing - `https://firebase.google.com/pricing`

[2]Expo - more on Expo in Section 2.4

[3]Electron is a framework for building cross-platform desktop applications with the use of JavaScript, HTML and CSS - more at `https://www.electronjs.org/`

worrying about browser differences, and spend more time on the actual application development. PouchDB is a free open-source project, written in JavaScript, with a quite huge community of 339 individual contributors. The project has more than fourteen thousand stars on GitHub and is quite popular.

- urql is yet another GraphQL client which provides similar features to Apollo GraphQL. What is more interesting is that it also provides offline support as part of its Graph-Cache package. It is fairly similar to other technologies, except that there is no decent documentation on how to use it properly in mobile applications. React is supported, and from my research, it should be possible to use it in React-Native applications. It is a project under active development, so more features and supported platforms can be expected in the future. For offline operations, it uses Normalized Caching instead of a database like IndexedDB or WebSQL. Normalized Caching is a process where denormalized JSON data is normalized back to their original form as they were stored in the database engine on the back-end. This way, the cache more resembles the database. Normalized cache also stores information about the relations between the types.

- AWS(Amazon Web Services) AppSync is a fully managed and paid service that provides GraphQL APIs and tools similar to the other projects, including offline support, real time synchronization and caching. Key advantage is its integration with other Amazon services such as Identity and Access Management, AWS Lambda[4], AWS DynamoDB and more. Also, AWS AppSync scales the GraphQL API engine automatically according to current traffic. AWS AppSync comes hand-in-hand with the Amplify Framework[5] which provides libraries for the data synchronization capabilities and more, for example, push notifications. The Framework supports all the main platforms like Native Android and iOS, React Native, Flutter, Ionic and works with other web frameworks too.

These previously mentioned solutions are very popular and a more complete comparison is shows in Table 5.1 provides comparison between them. From the table, it becomes obvious that although the libraries contain the necessary functionalities, it is quite challenging to use them due to lack of user-friendly documentation and examples. In the case of RxDB a React Native example application is available but is broken and non-functional, with attempts to fix it not being successful.

[4]AWS Lambda is a serverless compute service - `https://aws.amazon.com/lambda/`
[5]Amplify Framework Documentation - `https://docs.amplify.aws/`

Features	urql	RxDB	PouchDB	Offix DataStore	AWS AppSync
GraphQL	Yes	Yes	No	Yes	Yes
Schema generate	No	No	No	Yes	Yes
Conflict Resolution	No	Server Side	Server Side CouchDB	Server Side**	Server Side
Offline Storage	IndexedDB	IndexedDB SQLite	IndexedDB SQLite	IndexedDB	IndexedDB SQLite
Relationships models	Yes	Yes	Yes (CouchDB)	Yes	Yes
React support	Yes	Yes	Yes	Yes	Yes
React Native	No*	Yes*	Yes*	Yes*	Yes
RN Docs	No	Yes*	No	Yes	Yes
Flutter	No	No	No	No	Yes
Cordova	No	Yes	Yes	No	Yes
Native apps	No	No	No	No	Yes
Example applications	Yes*	Yes	No	Yes*	Yes

Table 5.1: Table comparing different libraries and their Features.
**It may be possible to use, but there is no official documentation or Example on how.*
***If used with the Graphback back-end.*

So far, the best solution at the time of research seems to be AWS AppSync. Their documentation is easy to read, many publicly accessible examples exist[6], and it supports multiple platforms and frameworks. The biggest drawback though, is that it is not a self-hosted solution and is tightly coupled to the AWS ecosystem, unable to be used without it.

5.1 IndexedDB

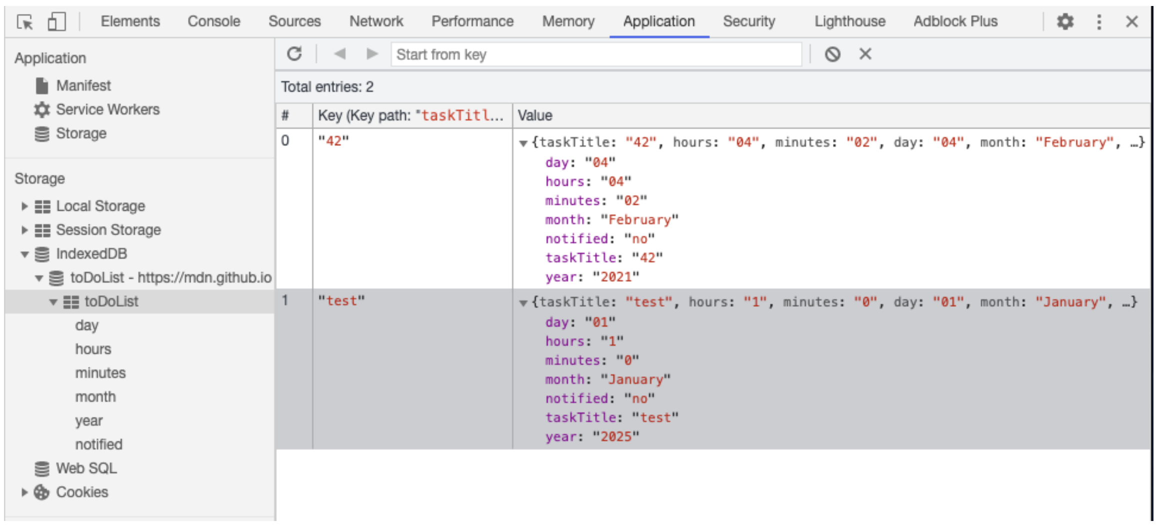
A significant stepping stone in Data Synchronization is adoption of Indexed Database API(commonly referred to as IndexedDB) as the main client storage for all data of the application inside the browser.

The first version of IndexedDB became a W3C Recommendation[7] in 2015. Currently, version 3.0 is in the stage of First Public Working Draft, as of March 11th, 2021, which means that the specification was published for review by the community, W3C members and other organizations.

IndexedDB at its core is a transactional database system, like many SQL database engines. However, IndexedDB is a JavaScript-based object-oriented database. IndexedDB stores and retrieves objects that are indexed with a key. Operations on the IndexedDB are performed asynchronously, to not keep blocking the applications while waiting for data to be fetched.

[6] AWS AppSync examples on Github - https://github.com/aws-samples

[7] W3C (World Wide Web Consortium) Recommendation is a stage of development where W3C can recommend the wide deployment of the specification in question and the specification becomes standard web technology. https://www.w3.org/2004/02/Process-20040205/tr.html



Total entries: 2

#	Key (Key path: "taskTitl…	Value
0	"42"	▾{taskTitle: "42", hours: "04", minutes: "02", day: "04", month: "February", …} day: "04" hours: "04" minutes: "02" month: "February" notified: "no" taskTitle: "42" year: "2021"
1	"test"	▾{taskTitle: "test", hours: "1", minutes: "0", day: "01", month: "January", …} day: "01" hours: "1" minutes: "0" month: "January" notified: "no" taskTitle: "test" year: "2025"

Figure 5.1: IndexedDB data structure example from example application by MDN Web Docs.

Figure 5.1 is an example of the structure in which IndexedDB stores data. In this example, the key is the name of the task and value is a JavaScript object containing all the details. A web pages could have multiple databases for different purposes.

Chapter 6

Conclusion

As part of this work, we have discovered that there are multiple libraries for Data Synchronization with both opened and closed source code, and that could be self-hosted or managed. But when it comes to the self-hosted part of the market, the libraries are no silver bullet solutions because they all come with their own specific problems and limitations.

Offix DataStore is quite a promising project, but given the early stages of development, it will still take some time before it is possible to use it in some more complex scenarios.

Among all the researched options, only AWS AppSync and Offix DataStore work well with some schema and code generators, which is also a very important aspect to consider. A developer should not be required to manually change the schema and code every time a small change is made to the data model.

For future development, the example application could be refactored to use the Offix DataStore instead of the deprecated Offix Client, but only when the library is in a state that would allow me to do so. The library shows a real potential in the self-hosted space, which with the rising interests in user data privacy will become more important for users with concerns about their data.

Bibliography

[1] Adobe I/O. *Update for Customers Using PhoneGap and PhoneGap Build* [online]. Medium.com, 2020 [cit. 2021-03-02]. Available at: https://blog.phonegap.com/cc701c77502c.

[2] Boduch, A. and Derks, R. *React and React Native: A complete hands-on guide to modern web and mobile development with React.js.* 3rd ed. Packt Publishing, 2020. ISBN 978-1-83921-114-0.

[3] Google Developers. *Announcing Flutter 2* [online]. Google Developers, 2021 [cit. 2021-03-03]. Available at: https://developers.googleblog.com/2021/03/announcing-flutter-2.html.

[4] Google Developers. *Cloud Firestore* [online]. Google Developers, 2021 [cit. 2021-03-04]. Available at: https://firebase.google.com/docs/firestore.

[5] IBM Cloud Education. *NoSQL Databases* [online]. IBM Cloud Education, 2019 [cit. 2021-03-03]. Available at: https://www.ibm.com/cloud/learn/nosql-databases.

[6] Mathur, A. and Chetty, M. Impact of User Characteristics on Attitudes TowardsAutomatic Mobile Application Updates. In: [online]. USENIX Association, 2017. Available at: https://www.usenix.org/system/files/conference/soups2017/soups2017-mathur.pdf.

[7] MongoDB Inc.. *How to Scale MongoDB* [online]. MongoDB Inc., 2021 [cit. 2021-03-03]. Available at: https://www.mongodb.com/basics/scaling.

[8] MongoDB Inc.. *NoSQL Explained* [online]. MongoDB Inc., 2021 [cit. 2021-03-03]. Available at: https://www.mongodb.com/nosql-explained.

[9] Poirier Ginter, M. *Using Node.js Express to Quickly Build a GraphQL Server* [online]. Snipcart inc., 2019 [cit. 2021-03-04]. Available at: https://snipcart.com/blog/graphql-nodejs-express-tutorial.

[10] RxBD Community. *A realtime Database for JavaScript Applications* [online]. RxBD Community, 2021 [cit. 2021-03-09]. Available at: https://rxdb.info.

[11] Solid IT GmbH. *Trend of Graph DBMS Popularity* [online]. Solid IT GmbH, 2021 [cit. 2021-03-03]. Available at: https://db-engines.com/en/ranking_trend/graph+dbms.

[12] Statcounter. *Desktop vs Mobile Market Share Worldwide* [online]. Statcounter, 2021 [cit. 2021-03-02]. Available at: https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#yearly-2010-2021.

[13] The GraphQL Foundation. *Queries and Mutations* [online]. The GraphQL Foundation, 2021 [cit. 2021-03-04]. Available at: https://graphql.org/learn/queries/.