MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Design and Implementation of a Mobile Application with Offline Support

Master's Thesis

JIŘÍ LOUN

Brno, Fall 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Design and Implementation of a Mobile Application with Offline Support

Master's Thesis

JIŘÍ LOUN

Advisor: RNDr. Pavel Novák

Department of Computer Systems and Communications

Brno, Fall 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jiří Loun

Advisor: RNDr. Pavel Novák

Acknowledgements

TBA

Abstract

This thesis documents the process of designing and implementing an enterprise mobile application with offline support in a real-world scenario with specific functional requirements to be fulfilled. By analyzing viable solutions and architectures, this thesis also serves as a reference for architects to be consulted when designing an optimal approach for a similar application.

Keywords

Mobile, React, React Native, Kotlin, Offline mode, Architecture, Synchronization

Contents

List of Tables

List of Figures

Introduction

The primary objective of this thesis is to ...

It consists of 4 chapters; the first chapter investigates the concept of offline-enabled mobile applications, their advantages, disadvantages, and specific challenges and architectural demands. The second chapter explores the approaches to be taken when designing an architecture and technology stack of such app - each approach is further documented, stating its strengths and weaknesses and practical scenarios in which each of them play their prime. Here the thesis not only focuses on different frameworks or languages, but also, the challenges introduced in the first chapter are confronted with different technical solutions and evaluated.

The third and fourth chapters focus on selecting the right approaches in different areas based on the functional and non-functional requirenments, implementation, testing, and other aspects of the practical execution of the project in the context of a real project for a real customer.

The end application conforms with the requirements and is deployed to production for the end users to work with.

Decribe motivation and purpose, business context ... TODO Tibi + nejaky clanek idealne? Why does the app solve the problem? What value does it bring?

1 Offline-enabled mobile applications

First, we need to take a look at offline-enabled enterprise mobile applications as a whole and identify the challenges they have to solve in addition to conventional mobile applications. By offline-enabled I mean an application that provides a significant part of its functionality without access to the server, while the extent of said functionality may vary based on a particular project and customer's needs.

What also tends to vary, and has a great influence on the project complexity, is the "extent" of offline capabilities. There may be applications that only need to display some data offline without modifying anything, or applications that need to survive only a couple of hours without synchronizing data with the server, but there may be applications that need to be backend-independent for days and then be able to synchronize the data with the server while running into as few conflicts as possible.

1.1 Concept

Mobile applications in the context of enterprise solutions always conform with the server-client schema — this thesis does not consider local-only applications as those have little to no use on the relevant market. In this server-client schema, there are two basic ideologies to pursue — server-heavy or client-heavy solution.

A server-heavy solution will rely largely on the backend capabilities. Any list filtering or sorting will be done through an endpoint, for example using a data search engine such as Elasticsearch[1]. The app will store as little state as possible on the client and will depend on server-managed state, settings, or data.

If this concept is deepened further in the context of web applications, whole pages can be rendered on the server and sent already partially prepared (pre-rendered, waiting to be hydrated — enriched by client-rendered functionality — on the client) to the client's browser.

1. https://www.elastic.co/elasticsearch

Such approach is called SSR (Server Side Rendering) and is currently on the rise with frameworks such as Next.js[2] — a framework which has gained a lot of popularity in the recent years, according to Francisco Moretti.

A client-heavy application, on the other hand, will rather utilize the client's resources to process or transform data instead of asking the server to do it. It will prefer sorting or filtering data in the browser or managing state locally — for example, by using a state-management library like Redux[3] or Zustand[4].

When considering any offline capabilities, the application must not be heavily dependent on the server and thus will go in the cleint-heavy direction. To accomodate the offline needs, the concept of the application will need to further deepen the client's independence in order to be able to retain its functionality even when the server is not responsive due to signal loss. The specific problems that one encounters while developing such an app are further explored in the next section.

1.2 Issues to be tackled

The first and the most obvious task is to make the application available offline as a whole. While it may seem trivial for certain platforms — e.g. installing an Android application from Google Play Store — it may be the first obstacle in other approaches, e.g. React PWA (Progressive Web Application). These specifics will be examined in the following chapter; at present, we turn to more conceptual issues.

1.2.1 Offline data availability

In order for the aplication to be able to work with data in offline mode, these data must be downloaded in advance and kept updated. This presents a unique challenge that is not dependent on platform or

2. https://nextjs.org/
3. https://redux.js.org/
4. https://github.com/pmndrs/zustand

framework choice — it is, generally, not a good practice to download all data that could possibly be accessed in the application — that would practically mean data mirroring of whole database or a large part of it. That might not even be possible due to the database size. It is, therefore, quite imperative to try to limit the amount of data that should be available offline from the business logic side.

As an example, let's imagine a remote-synchronized enterprise calendar application for planning meetings. Meetings can be input from a device and synchronized to a server. All devices that are subscribed to a certain calendar (e.g. meetings for project X) have it automatically synced with the server to display events submitted from other devices. It might very well suffice to automatically download only the current week or month for offline purposes as there is probably low business value in viewing meeting plans several months in advance or in history.

1.2.2 Partial changes management

A related topic to the previous section is partial changes management — a concept of recording changes of data, not entire snapshots.

The use-case here is updating the local data to match the server state — the longer the application has been in offline mode, the higher the probability of the data being out of date. Specifically, the probability of the data being out of date after a certain period of time follows the Poisson's distribution[2]:

$$P(X = k) = \frac{\lambda^k}{k!} e^{-\lambda} \tag{1.1}$$

where λ is expected rate of occurence and k is the number of occurences. Let's assume that on average during working hours, each instance receives an update every two hours. That is 4 requests during a whole working day. Therefore $\lambda = 4$. The probability that the instance is still up to date the next day (after 8 working hours), in other words, $k = 0$, is approximately 1.83%.

Therefore, the local data should be periodically or manually updated to minimize the occurence of synchronization conflicts. The

reason why the system should support partial changes mangement is that if there is none, the only option of a local data update is to bulk-download all data, as if there were none stored, and overwrite them.

There are two layers of this principle that a system can implement:

Tracking instances

The first layer is keeping track of whole objects (instances) that have been changed since a particular point in time — essentially, having an `updated` attribute in each instance that holds the timestamp of the last change of its attributes. In this case, when the client is asking the server for fresh data to update its local copy, it can only request fresh data of instances that were updated since the last data download.

There also has to be a check for any new relevant instances or any deleted ones - those must be deleted (or soft-deleted[5]) from the local data on the client.

Going back to the meeting calendar application example, the client would like to update its local list of planned meetings with their details. There would have to be an endpoint, say `POST /meetings/data/update` and the structure of the body of such a request could, for example, look like this:

```
{
  "since": ISO string,
  "knownIds": uuid[],
}
```

where `since` is the timestamp of last data update on the device (every newer change needs to be provided) and `knownIds` is an array of IDs of planned meetings that the device currently tracks.

The response to such request could have this structure:

```
{
  "added": Meeting[],
```

5. Soft-delete is a technique when an instance is marked as deleted instead of being deleted.

```
  "deleted": uuid[],
  "updated": Meeting[],
}
```

where in the `updated` and `deleted` parts, only the instances from the sent `knownIds` are considered by the server.

Tracking attributes

Going even deeper in granularity, each attribute can have its timestamp and thus the response to the local-data-updating request may only contain changes in attributes and does not have to send the whole data object. While timestamping every attribute might sound like an overkill, as will be demonstrated later, in a complex offline-ready system, it may serve multiple purposes and therefore might be worth implementing.

Extending the simple request-response schema from the previous section, the only change would be that `updated` would be of type `MeetingPartial` - containing only meeting ID and attributes that have been changed.

1.2.3 Local data management

In a traditional client-server based application, the client should not persist any data that might be volatile. The aim is to fetch a fresh copy of data each time the user displays a certain page for them to work with the latest version of data[6]. That is especially important for data updates — in case there is any form of version control (even a simple one — every instance has a current version v and every PUT request must have version $v + 1$, otherwise the server refuses to process the request due to version inconsistency), the data freshness might be the differentiator between having the request processed or not.

After working with the data and possibly mutating them, they are sent to the server to process them and after receiving a positive

6. Here I am ommiting the realm of real-time applications as those are rather specific.

response, the data is refreshed. Hence, the local copy of data is fresh once again and the old copy is thrown away - as an example, let's imagine a `updated` attribute — one that stores the date and time of the last change that has been done to the instance. Updating the value of this attribute will only be done on the server, but the client definitely wants to see this attribute updated after submitting a mutation to the entity.

In the offline world, there is no fresh data and there is no server. All mutations have to be done on the local data copy, including attributes like `updated`. The server interaction needs to be faked by the application itself — which, in reality, means implementing a lot of server functionality on the client, which represents the client thickness.

1.2.4 Server synchronization

Synchronizing the changes ade on the client to the server and managing potential conflicts is arguably one of the most difficult tasks to tackle in the field of offline-enabled applications. The task is to synchronize all changes made in offline mode to the server when coming back online as seamlessly as possible.

The reason why conflicts arise is that the system, and therefore (generally) any data in it can be accessed and mutated by other users, while one or more users are making changes in offline mode. There may be two general approaches:

State-based synchronization

All the mutating changes done in offline mode affect directly the local copy of data while marking it *dirty* — when switching back to online, every dirty instance is synchronized to the server in one mutation - one PUT request. Newly created (POST) instances that have later been updated are synchronized as one POST with all the updates already put in. Updating an instance and then deleting it means only the DELETE request is synchronized and the updates are thrown away.

Operation-based synchronization

Each and every update is propagated to the server. That means, there is a layer that tracks all the requests that have been done (preferably with a timestamp) and when the device comes back online, transmits them all for the server to deal with each change in sequence[7]. This approach may lead to a lot more conflicts to be resolved — every request might generate a conflict, so more requests means more potential conflicts.

There is one more issue that arises in this approach: when creating a new instance, it is assigned an ID (locally) for unique identification. Later, but still in offline, the instance might be updated — this change will be recorded as a PUT with the generated ID. When the synchronization to server starts and the POST is transmitted, the server may assign it an ID based on its ID-generation strategy and this ID will probably be different than the locally generated one. The subsequent PUT would then have an invalid ID. To resolve the issue, the client needs to keep a relation of local and server IDs, much like a translation table, and use it to modify any subsequent calls that reference the instance.

Despite these limitations and added complexity, there are several reasons to prefer this approach:

1. Auditability: each change is recorded, so there is full control over who did what and when it has been done.
2. Granularity: when one change out of ten causes a conflict, in the operation-based approach it is clear which one it is, while the state-based approach will diffuse it.
3. Business logic: it may not be possible to implement the state-based approach in some cases due to functional requirements — when a change of an attribute has its meaning. Imagine a package delivery app where the pacakge has a state (in warehouse, in delivery, delivered). The courier's phone is currently not connected

7. Optimally, the server also accepts the client's timestamp and records the change to have been done in the time of the timestamp, not the request time. Therefore, the real timing of changes, creation, or deletion is also recorded.

to the interet as he picks up the package from a warehouse and delivers it to the customer. It is only after that when their phone finally connects to the internet. In state-based approach, the package would go straight from *inWarehouse* state to *delivered* state — which is probably not the expected behaviour.

1.2.5 Offline authentication, authorization

Application usability strongly relies on the user remaining authenticated while offline — they cannot be automatically signed out when their token expires. Either the token-checking logic has to be suspended while offline, or the server response to such request must be mocked to a succesful call.

But, remaining signed into the application while offline may cause a security issue — the local data can be sensitive and without authorization, an intruder that has taken over the device, even after a long time trying, can see the data unauthorized, provided that the device remains offline. This risk can be mitigated by, for example, signing the user out after a specific amount of time in offline, but that has to be agreed-upon from the business side.

Yet there is still an additional issue. Even when the user is signed out from the application, the local data is still present in the storage of the device. In case this poses a considerable security issue in the customer's context, the data must be encrypted by the application.

2 Technical approaches

2.1 Frameworks

list of FE approaches to take with their (dis)advantages

2.1.1 React-Native

2.1.2 React PWA

2.1.3 Native approaches

Kotlin

Flutter

2.1.4 Others

2.2 Local data management

approaches to tackle the local data management

2.2.1 Local database

fake database

2.2.2 Request intercepting, processing

fake server

2.2.3 TBA?

2.3 Synchronization

2.3.1 Mitigation

2.3.2 Duplication

2.3.3 Atribute timestamping

2.3.4 CRDT

2.3.5 Advanced synchronization management

(approaches based on advanced methods like GIT etc)

3 Implementation

3.1 Functional and non-functional requirements

3.2 Architecture, technology stack

3.3 Service Worker

3.4 Axios Interceptor

3.5 Local data management

3.6 Synchronization

4 Issues and future work

Conclusion

Bibliography

1. MORETTI, Francisco. Next.js Revolution — The framework popularity rises in 2023 [online]. 2023 [visited on 2025-09-27]. Available from: https://medium.com/@franciscomoretti/next-js-revolution-the-framework-popularity-rises-in-2023-a8ebc52d673a.
2. ZHAO, Jing; ZHANG, Fan; ZHAO, Chao; WU, Gang; WANG, Haitao; CAO, Xinyu. The Properties and Application of Poisson Distribution. 2020, vol. 1550, no. 3, p. 032109. Available from DOI: 10.1088/1742-6596/1550/3/032109.

A Electronic attachments