MASARYK UNIVERSITY
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

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 design and implemtent an offline-ready enterprise mobile client application. The secondary objective is to provide a comprehensive guide through designing such an applictation while considering different requirements. This guide features a list of available technological solutions with their advantages and setbacks while addressing the specific issues that arise in this architecture. The guide will be evaluated against a set of real-world functional and non-functional requirements and a real application that adheres to the result of that evaluation will be designed and implemented to be used by a customer as an enterprise solution.

The thesis 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 requirements, implementation of a mobile frontend client, testing, and other aspects of the practical execution of the project in the context of a real solution 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, it is needed 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 is meant an application that provides a significant part of its functionality without access to a 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[1].

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 occurrence and k is the number of occurrences. Let's assume that on average an instance of an entity is changed once every 2 hours during working hours by some user. That is 4 changes 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 occurrence of synchronization conflicts. The

reason why the system should support partial changes management 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 the realm of real-time applications are ommited 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

This chapter explores different technical approaches to employ when designing an enterprise solution in the context outlined in the previous chapter.

2.1 Frameworks

First, there is the crucial decision over the base architecture of the application, that is, the language and framework to be used. This choice has a lot of implications for later design choices — for example, lack of certain features in certain frameworks may mean resctricted choice of solutions for a particular challenge.

Since the topic at hand is mobile applications, naturally the first thing that comes to mind are mobile-specific (mobile-only) languages and frameworks. These can be split into native approaches such as Kotlin or Swift, and cross-platform ones such as React Native or Flutter.

But there has been a strong tendency to favor browser-based solutions over native ones in recent history, as Jon Henshaw[3] comments. He argues that Google has been one of the biggest driving forces on the market and that the sheer market share of their browser, Google Chrome, can in part be attributed to this transformation. The architecture in question is SPA (single page application), specifically in form of PWAs (progressive web applications).

Tools that Henshaw mentions, like Google Meet, YouTube or Canva, are only a fraction of the sheer quantity of applications that use this web-based architecture. Spotify, Netflix, or applications like BandLab[1] — a fully-featured in-browser music creation platform. After all, even Microsoft's office suite has been migrated to the browser with the Office 365 and as Bill Doll[4] explains, Microsoft has been heavily investing to make their office suite web-first for quite a few years now, even prioritising the web variant over the native one for new feature releases.

---

1. https://www.bandlab.com/

2.1.1 Native approaches

The main argument to use native solutions is the closeness to the operating system. That means, the fewer layers the better performance, smoother UX (user experience) and better system integration. But it may also bring greater complexity and management.

Advantages

Performance is among the most prominent reasons to go native. Philipp Horn; Rituparna Bose; Christian Becker[5] performed a qualitative study that compared Native Applications with their web-based counterparts, finding that there is a statistically significant advantage of native apps in not only performance (startup time, responsiveness), but also power consumption, resource consumption (such as CPU[2] or memory) and network traffic. Therefore, when the application is performance-demanding, the native approach should not be overlooked.

There is a related advantage of native applications that enhances the user's perception of performance — the UI (user interface) / UX integration into the system. Since the applications are native, they respect the animations, navigation, and overall user experience. It will be very well optimized for the device and may even be coupled with haptic feedback, system shortcuts or widgets.

The last notable strength of this architecture that should be discussed is operating-system-level integration. Native apps can communicate with the device's hardware and core features immesurably better than a browser application will. That includes all the sensors that a mobile device has — camera, Bluetooth, NFC (Near Field Communication), Accelerometer/Gyroscope, ambient light, temperature sensors, and a lot more. All of these can be directly integrated into a native app.

This area is not only limited to hardware itself, there is the OS (Operating System) layer as well — managing foreground and back-

---

2. Cetral Processing Unit

ground jobs, having full control over OS notification channels, battery optimizations, or native sharing options and integration with other native applications. To conclude, when the application needs to have a deep hardware or operating level integration, the native approach plays its prime.

Disadvantages

On the other hand, there certainly are disadvantages to this approach. The greatest one being the fact that when multiple operating systems need to be supported, there have to be multiple codebases in use — each for one platform.

Typically, that would mean Kotlin for Android support and Swift for iOS support. There is quite a limited extent to which certain logic can be shared between different languages and frameworks. That means a lot more cost to develop and maintain two application versions instead of one. The applications also need to be published and then updated by respective app stores, so the deployment process can be a little more difficult.

The other disadvantage that should be talked about comes from the very same reason that is mentioned earlier as an advantage — the closeness to the OS and the level of control over it, or in other words, the absence of abstracting layers between the application and the OS. Less abstraction leads to heavier reliance on specific OS versions and flavors.

For example, every Android version on every phone brand differs slightly and to ensure the app compliance with a wide variety of supported devices, the testing needs to be extensive and the development might have to include a lot more specific case handling.

Kotlin

What is often referenced as Kotlin is in fact Kotlin + Android SDK (software development kit) + Jetpack Compose. Kotlin[3] is the pro-

---

3. https://kotlinlang.org/

gramming language developed by JetBrains. It was designed to be a modern yet mature programming language that takes a lot of inspiration from languages like Java, C# or Scala[6]. One of its strong points is that it is fully interoperable with Java, which makes migration as painless as possible.

Google states that Kotlin provides better expressiveness, conciseness and safety over Java, which are the main reasons of why in 2019, Google annouced that Android development will be Kotlin-first[7]. According to the Stack Overflow Developer Survey from 2024[8], developers enjoy working with Kotlin more than with Java, which is shown by the admiration index where Kotlin surpasses Java by almost 15% (47.6% for Java, 60.9% for Kotlin).

Calling the whole stack just 'Kotlin' is the norm in the developer community not only because of it being the official language chosen by Google, but also because the main usage of the Kotlin language is Android development, as can be seen in the JetBrains 2021 Developer Ecosystem Report[9].

To comment on the other parts of the Android development stack, the Android SDK is a layer between the Android OS and the Kotlin code that enables developers to communicate with system resources in a form of an API (application programming interface) together with build tools or an emulator for testing. Jetpack Compose is a UI toolkit for Android applications. It is the recommended way to design and build the UI by Google[10].

Swift

Swift[4] is for iOS what Kotlin is for Android. Developed by Apple, designed to replace Objective-C, much like Kotlin was designed to replace Java, it is the official language for its respective OS. Its main focus is on speed, expressiveness and safety while providing interoperability with C and C++. According to the aforementioned Stack Overflow Developer Survey[8], Swift is even more admired amongst

---

4. https://www.swift.org/

developers with 63.3% admiration index, while virtually obliterating Objective-C with its 26.3%.

In the technology stack, similarly to the Android ecosystem, Apple SDK provides the above-OS layer, and SwiftUI is the counterpart to Jetpack Compose.

Talking about comparisons between Kotlin and Swift, performance is the first that might come to mind. according to a set of benchmarks[5], Kotlin seems to generally complete tasks faster when coupled with JVM (Java Virtual Machine). There are other technical differences such as Kotlin's JVM-powered Garbage Collection Approach (GCA) versus Swift's Automatic References Counting (ARC) for memory management[11] and other caveats, yet there is not much sense in such comparisons as the languages are designed for completely different environments.

There is one direct comparison, however, that is to be made - the syntax of the languages, revealing their expressiveness. Two code snippets of a simple class follow, the former in Swift and the latter in Kotlin. As expressiveness is subjective, it is up to the reader to decide the winner.

Listing 2.1: Swift example

```swift
class Pet {
    var name: String
    var age: Int
    var sound: String?

    init(name: String, age: Int, sound: String?
      = nil) {
        self.name = name
        self.age = age
        self.sound = sound
    }

    func getSoundMessage() -> String {
```

---

5. https://programming-language-benchmarks.vercel.app/swift-vs-kotlin

```
        return (sound != nil) ? "\(sound!)!" : "
            I do not make any sound..."
    }

    func printGreeting() {
        let soundMessage = getSoundMessage()
        print("Hello, my name is \(name) and I
            am \(age) years old. \(soundMessage)")
    }
}
```

Listing 2.2: Kotlin example

```
class Pet(
    var name: String,
    var age: Int,
    var sound: String? = null
) {

    fun getSoundMessage(): String = sound?.plus(
        "!") ?: "I do not make any sound..."


    fun printGreeting() {
        val soundMessage = getSoundMessage()
        println("Hello, my name is $name and I
            am $age years old. $soundMessage")
    }
}
```

2.1.2 Cross-platform approaches

As has been highlighted in the last section, one of the main drawbacks of the native approaches is that when multiple operating systems must be supported, the same functionality has to be implemented in multiple codebases. That is a major reason not to elect them as the ideal solution — and that is where multiplatform approaches come

into play. There are two main options to choose — Flutter or React Native.

A layer of isolation between the application and the OS is introduced that abstracts the OS-specific differences such as system APIs or other SDK functionality — which carries with it its pros and cons.

Advantages

The most significant advantage is having a single codebase, which reduces costs dramatically. The layer of abstraction over the OS ensures that no matter the specificity of the OS, there will be common functionality that is expected to work on all of them. This layer communicates directly with the OS itself, so harware availability, like access to camera, filesystem, or sensors, is still supported.

Disadvantages

Despite this approach solving the biggest problems with native solutions, it is far from perfect. According to research done by Ihor Demedyuk; Nazar Tsybulskyi[12], Flutter and especially React Native have poorer performance when compared to their native counterparts.

The abstraction over more operating systems also means the control over specific functionality may not be so precise. For example, there might be some camera controls or sensors available only on one OS that may get dropped during the abstraction or must be handled with platform-specific code. New features may also come with a delay as they have to be layered over first instead of just implementing the system API in Kotlin or Swift.

Flutter

Flutter[6] is a framework developed by Google that can be compiled to native code and be used on Android, iOS, desktop or web[13]. It uses its own rendering engine developed with Skia (a 2D rendering

---

6. https://flutter.dev/

engine) and Impeller (iOS rendering engine), which means that the UI will be uniform on all platforms regardless of the platform-specific UI or animations. While that may improve brand consistency, it may suffer from OS inconsistencies.

Flutter uses Dart[7] as its programming language. Developed by Google, Dart is not a broadly used language — in the 2024 Stack Overflow Developer Survey[8], it reached only 6% in popularity and 6.2% in desirability. Its admirability index is not anything special either with the score of 55%. It is a strongly typed, object oriented language that can be compiled to machine code, JavaScript, or WebAssembly.

In the aforementioned benchmark, Flutter achieves significantly higher performance in intensive tests. If performance is an important factor, but native approach is not to be considered, Flutter is the better option.

Flutter uses a tree-like widget structure for its UI — a widget is a single building block for any UI element. It resembles native-like syntax, meaning Kotlin or Swift developers should find it approachable. A short code snippet follows that illustrates the widget declaration — a simple clickable User Card that displays the user's name and profile picture; clicking it navigates the user to the user screen.

Listing 2.3: Flutter example — User Card

```
class UserCard extends StatelessWidget {
  final User user;

  const UserCard({Key? key, required this.user})
    : super(key: key);

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () => Navigator.pushNamed(context,
        '/user'),
      child: Card(
        child: ListTile(
```

---

7. https://dart.dev/

```
        title: Text(user.name),
        trailing: CircleAvatar(
          backgroundImage: NetworkImage(user.
            profilePictureUrl),
          radius: 16,
        ),
      )
    ),
  );
}
}
```

React Native

React Native[8] is a cross-platform framework developed by Meta that can be used to develop mobile applications for both iOS and Android. It uses JavaScript (JS) or TypeScript (TS) as its programming language. This is one of the biggest advantages over Flutter as those languages are very popular, meaning there is a huge developer base and an abundance of tutorials, guides and forum pages to help the developer when encountering an issue.

Specifically, according to the Stack Overflow 2024 Developer Survey[8], Javascript is the most popular language with 62.3% popularity while TypeScript is fifth overall with 38.5% populartiy. They are also desired (39.8% — second place overall — for JS and 33.8% — fifth place overall — for TS) and TypeScript is also admired with 69.5% of admirability (JavaScript falls short at 58.3%).

The JavaScript/TypeScript popularity also means there is a vast universe of third party libraries developed for them, meaning that when in need of a specific library, e.g. a barcode scanner, there is a good chance someone has built that library and it can be used in the project as a plug-and-play component.

React Native does not have a custom rendering engine like Flutter does, its code is rendered as native UI elements. This connection

---

8. https://reactnative.dev/

between the JavaScript and the Native world is possible thanks to a "bridge"[14]. This bridge can also be used to write platform-specific native code, for example in Kotlin. This unique feature allows React Native applications to overcome the abstraction layer when needed, which may be very powerful.

React Native uses components as building blocks, much like Flutter uses widgets. They, much like React, are JSX/TSX functions, therefore written in markup style. To illustrate the syntax difference compared to Flutter, a JSX component of a User Card follows:

Listing 2.4: React Native example — User Card

```
export default function UserCard({ user }) {
  const navigation = useNavigation();

  return (
    <TouchableOpacity onPress={() => navigation.
      navigate("User")}>
      <Card>
        <Card.Title
          title={user.name}
          right={() => (
            <Image
              source={{ uri: user.
                profilePictureUrl }}
              style={{ borderRadius: 16 }}
            />
          )}
        />
      </Card>
    </TouchableOpacity>
  );
}
```

19

2.1.3 React PWA

A React PWA (Progressive Web Application)[9]

2.1.4 nejakej vytazek lol, proste conclusion frameworku nebo tak

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)

---

9. TODO

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 Testing, 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`.

3. HENSHAW, Jon. *Why Web Apps Are Quietly Replacing Native Apps on Phones and Computers*. Coywolf News, 2024. Available also from: `https : / / coywolf . com / news / productivity / why - web - apps-are-quietly-replacing-native-apps-on-phones-and- computers/`. Accessed: 2025-10-20.

4. DOLL, Bill. *Why You Should Use Office for the Web*. Microsoft 365 Blog, 2019. Available also from: `https : / / techcommunity . micr osoft . com / blog / microsoft _ 365blog / why - you - should - use - office-for-the-web/567060`. Accessed: 2025-10-20.

5. HORN, Philipp; BOSE, Rituparna; BECKER, Christian. Native vs Web Apps: Comparing the Energy Consumption and Performance of Android Apps and their Web Counterparts. *arXiv preprint arXiv:2308.16734*. 2023. Available also from: `https : / / arxiv.org/abs/2308.16734`. Accessed: 2025-10-20.

6. GOOGLE LLC – ANDROID DEVELOPERS. *Kotlin and Android* [`https : / / developer . android . com / kotlin / overview`]. 2024. Accessed: 2025-10-23.

7. GOOGLE LLC – ANDROID DEVELOPERS. *Android's Kotlin-first approach* [`https : / / developer . android . com / kotlin / first`]. 2024. Last updated June 27, 2024. Accessed: 2025-10-23.

8. STACK OVERFLOW. *Stack Overflow Developer Survey 2024*. 2024. Available also from: `https://survey.stackoverflow.co/2024/`. Accessed: 2025-10-23.

9. JETBRAINS. *The State of Developer Ecosystem 2024* [https://www.jetbrains.com/lp/devecosystem-2024/]. 2024. Accessed: 2025-10-23.

10. GOOGLE LLC – ANDROID DEVELOPERS. *Jetpack Compose — Android's recommended modern UI toolkit* [https://developer.android.com/compose]. 2025. Accessed: 2025-10-23.

11. JAFTON. *Kotlin vs. Swift: How Are They Different?* Jafton, 2025. Available also from: https://www.jafton.com/insights/kotlin-vs-swift. Accessed: 2025-10-27.

12. DEMEDYUK, Ihor; TSYBULSKYI, Nazar. *Flutter vs Native vs React-Native: Examining Performance* [online]. inVerita, 2020. [visited on 2025-10-20]. Available from: https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980.

13. FLUTTER TEAM. *Flutter FAQ* [online]. 2025. [visited on 2025-10-27]. Available from: https://docs.flutter.dev/resources/faq.

14. BIRD, Shane. *React Native: All You Need To Know* [online]. Blott Studio, 2022. [visited on 2025-10-27]. Available from: https://www.blott.com/blog/post/react-native-all-you-need-to-know.

A Electronic attachments