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 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 is 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

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.

not connected 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 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 package 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 of the system they are running on. They 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 significantly better than a browser application ever 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. Their data are presented through respective system APIs in (more or less) raw form. There is little abstraction and maximum control.

2. Cetral Processing Unit

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

Additionaly, there may also be a need to support Windows mobile devices as well, not even mentioning Ubuntu mobile and other alternatives.

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 a programming 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] for iOS is 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

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

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 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 superior one.

Listing 2.1: Swift example

```
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
    }
```

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

```
    func getSoundMessage() -> String {
        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 on the market 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.

Also, the app still needs to be distributed over multiple system app stores, which may cause delay in delivering new versions of the software.

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 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 performance 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
```

6. https://flutter.dev/
7. https://dart.dev/

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

8. https://reactnative.dev/

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 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 in 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>
```

```
    );
}
```

2.1.3 React PWA

React.js[9] is a powerful JavaScript frontend library/framework used for building modern user interfaces. Developed by Facebook/Meta and published in 2013[15], it is by far the most popular frontend framework at 39.5%, followed by Next.js[10] at 17.9%, which is a web framework built on top of React. It is also the most desired at 33.4% by a considerable margin and is reasonably admired at 62.2%, according to the Stack Overflow Developer Survey 2024[8].

From these numbers, it is safe to say that React is the single most used and important frontend framework on the current market. That means there is a massive amount of packages to be used for development in React and a multitude of guides, forums and other content about the framework which serves as an invaluable source of assistance during development. There is also a remarkable number of React developers on the market.

React uses JavaScript or TypeScript as the programming language (pros and cons of this choice are mentioned in the previous section about React Native), conceptually splitting the UI into singular components written in JSX (or TSX for TypeScript projects) syntax that serves as a JavaScript extension to write markup code — similar to HTML (Hypertext Markup Language), but enriched with JS elements — inside of JS functions[16]. For styling purposes, pure CSS (Cascading Style Sheets[11]) or any of its frameworks like Tailwind[12] may be used.

An example of the (by now well-known) User Card follows:

Listing 2.5: React + Tailwind example — User Card

9. https://react.dev/
10. https://nextjs.org/
11. https://developer.mozilla.org/en-US/docs/Web/CSS
12. https://tailwindcss.com/

```
export default function UserCard({ user }: Props
    ) {
  const navigate = useNavigate();
  return (
    <div
      className="flex␣flex-row␣justify-between␣
         rounded-xl␣shadow-lg␣p-4"
      onClick={() => navigate("/user")}
    >
      <span className="text-lg␣font-semibold">{
         user.name}</span>
      <img src={user.profilePictureUrl}
         className="rounded-full" />
    </div>
  );
}
```

PWA

A PWA (Progressive Web Application)[13] is a web application built as a capable standalone installable application enhanced with modern browser system APIs and offline capabilities[17].

A PWA can be installed straight from the browser through a context menu, meaning there is no need to handle app store publication or different builds for different platforms. There is only a singular codebase and a singular deployment — the "browser one" — that also provides the option to install the application on any platform, transforming it to a natively-feeling one. The updates are also managed fully in the application and are distributed from the server to all clients. That brings the option for the developer to always automatically update the client as soon as an update arrives, eliminating the risk of users using out-of-date applictation versions.

If needed, there is also a packaging option called PWABuilder[14] that can produce a application package that can be distributed to

13. https://web.dev/explore/progressive-web-apps
14. https://pwabuilder.com/

several app stores like Google Play Store, but using that option means losing the updating advantages described in the previous paragraph.

An installed PWA can be pinned to a mobile home screen, PC desktop, or a taskbar. It can be managed inside the OS, set up to use push notifications, registered to accept content from other system applications, set up to override browser keyboard shortcuts, or chosen to be the default application for opening files of certain types.

A PWA also makes web applications offline-ready. It has become a user-expected standard in the last couple of years for applications to provide some form of content even when offline. In the traditional Web Application context, that is not possible — the web application frontend itself is provided to the browser from the server, and when the request to load a page fails (because of network issues or server unavailability), the browser just displays an error page. A PWA enables advanced caching strategies handled by service workers.

A Service Worker is a middleware residing in the browser that can act as a network proxy between the client and the server[18]. Being a browser API running in the background, it practically enables offline mode or push notifications. The service worker has a developer-specified scope of network requests that it handles. Depending on the strategy chosen for each section of endpoints, it can handle them accordingly — serving fresh or cached, serving a custom-made data, or, for example, serving from cahe and fetching a fresh version of data in the background and updating the cache.

Advantages and Disadvantages

When weighing pros and cons of progressive web applications, one can view it is the polar opposite of the native approach — while the native solution is as close to the hardware and OS, this approach is as abstracted as possible, layering over the OS with not only a cross-platform layer over the system APIs like Flutter does, but a whole browser layer over it as well.

There is a certain level of advantages coming from this fact — with the solution practically detached from the OS, the developer only

needs to care about the browser capabilities. Features like camera support include the enumeration of front or back facing cameras, abstracting from PC webcams to multiple lenses on smartphones. Another example may be a file picker, handled by the browser, bringing up the correct dialog or window on any system and handling the file upload itself.

The browser abstraction layer also means that the PWA can only do things that the underlying browser can — opposite to native, the browser cannot influence the battery management, reach sensors like fingerprint sensor, WiFi details or any health-related sensor data.

The extent to which other system data or functionality is available is also restricted. Bringing up the camera once again, the in-browser camera capabilities are limited by the WebRTC[15] protocol stack — the camera quality is typically heavily reduced and its raw output is not processed by the OS like it would be in the native camera applictation on the device. That may lead to user frustration — why is the quality so much worse compared to taking the photo outside of the application when the harware is the same?

All the abstraction over the OS also means that the performance will be poorer when compared to native solutions when doing resource-intensive tasks. Yet the PWA may try to tackle this using smart caching strategies.

2.1.4 Conclusion

There are a few clear takeaways from the selected approaches to be taken when choosing the main application framework:

1. When performance in resource-intensive tasks is crucial, there is no better solution than native. Yet for “normal” usage, which is the expected one when talking about the context of enterprise mobile applications, the user may not feel the speed difference.

15. https://webrtc.org/

2. When the application's focus is on utilizing the device's hardware capabilities like NFC or heart-rate sensor, the correct choice is also to go native.

3. If possible and in accordance with the functional requirements, when needing to support more operation systems, going cross-platform may save a lot of resources thanks to only maintaining a single codebase.

4. When there is an option to go the PWA route, it should always be considered as it may be the cheapest and smoothest one thanks to a large developer base and a plethora of libraries to be used to enhance the application.

5. PWA is the easiest to distribute, fastest to install and most consistent over multiple platforms. But being the most abstracted one, precise access to various system resources is restricted.

2.2 Local data management

As chapter one explains, keeping track of local data and its changes in offline mode is a crucial issue to tackle. There are two magnitudes to be chosen for resolving this problem — faking just the server or also faking a database.

2.2.1 Request intercepting and processing

The first option is to build a custom proxy-like layer serving as a form of middleware between the client and the server. This layer intercepts all the mutating requests. The data getting (GET) requests are handled separately, for example in the form of cache. The POST, PUT, PATCH, and DELETE requests are stored as pending operations and a success state (200, 201, etc.) is returned to the client as if the request was successfully processed by the server. Later all the pending operations are synchronized one by one, in order, with the server.

Client validation

This *optimistic* UI update strategy also means that the data validation must always be done on the client and data consistency must be enforced there — there is no server to handle it, the client needs to ensure that the requests have the highest possible probability of succeding once they are actually transmitted throught the network and processed by the server.

Importance of this principle grows with any inter-endpoint or stateful dependencies. As an example, let's imagine an entity with several states and a lot of attributes. In each state, a certain subset of attributes is displayed to be filled. A change of state of an instance requires the instance (or its related/dependent entitites) to be filled with a valid values for the currently displayed set of attributes. The state change opens another subset of data to be filled, depending on the values of the previously submitted values.

That is a *dependency path* — a sequence of unique steps dependent on each other. If the user fills the first step incorrectly and the client fails to enforce the validity to the same extent that the server eventually will, each subsequent update to this instance must be thrown away during synchronization. As the validity of the first step is compromised, it is effectively compromising all the following steps — the first state change was not valid and therefore all subsequent dependent actions will not be either.

Incorporating mutations into local data

Assuming the validation is handled correctly, there is still one aspect that needs to be talked about — displaying the mutated data correctly. Earlier in this section, it was stated that the cached version of data is used for presentation. That fails as soon as there is a need to display mutated (updated) data in the UI — the pending operations need to be incorporated into the data.

The solution is to proxy also the GET requests and include any pending changes into the data presented by the client. As each entity is identified by its name (and therefore a specific endpoint group)

and every instance is identified by its ID, it effectively means fetching the instance data from the cache and searching through the pending mutations to filter those that affect this very instance — in other words, searching for a match in endpoint name and instance ID. All found mutations are then applied to the fetched (cached) data inside the middleware and presented to the client.

This intercepting approach fails to scale well for request-frequent or data-intensive applications as each GET request must be heavily processed and a structure of pending requests searched through repetitively. But it excels at dependent chains of requests where time and state is crucial. The upside is also that the client does not have to know about being offline at all as the middleware acts as an optimistic server — the client then fetches and mutates data as if the device was online and all is handled by the intercepting layer.

POST - PUT chain

In the previous section, an assumption about each instance being uniquely identified by an ID was made. There is one edge case that invalidates this assumption — creating an instance while offline. Traditionally, the server assigns IDs to new instances in a way that respects the integrity constraints of the database. The client, however, cannot reliably assign IDs on its own and synchronizing the POST request afterwards will lead to the ID being overwritten by the server, rendering all potential subsequent PUT/PATCH/DELETE requests to the same entity invalid as the ID would not match.

The simplest option is to prohibit the POST-PUT chain completely. If it makes sense in the context of a particular application where the vast majority of requests are mutating server data, the few use-cases that are creating and subsequently updating an instance may be transformed to simply remove the initial POST from the pending operations and creating an entirely new one. This approach is the simplest one, but when creation/update timestamping is crucial, it isnot a viable one.

The second option is to generate the ID on POST on the client and later during the synchronization map the client-generated ID to the

real server-assigned one and then alter all subsequent calls to this instance to respect the correct ID.

Generally, if the vast majority of functionality includes creating new instances and potenitially subsequently mutating them, this approach is not ideal and may not scale well; but when the majority is about stateful and dependent mutations, it has the upper hand.

2.2.2 Local database

The second, more robust, scalable, but also complex and demanding solution is to manage a local database on the client. The support for local databases is present in all frameworks mentioned in this thesis — from the Room database in Kotlin[16] to the browser's IndexedDB for PWAs[17].

The local database schema needs to closely mirror the server one and must implement the same integrity constraints, keys, or unique constraints to mitigate synchronization conflicts. The validation needs on the client are also as relevant in this approach as in the last one. The relevant data is fetched and stored to the local database while online and then, when offline, the application only works with the local database to access and mutate data, no requests are attempted. Relevant instances may be retrieved, created, updated or deleted from the database.

During the synchronization, the local tables must be merged with the server ones, writing through the data changes that occured while offline. The changed/created/deleted instances from the offline session would be flagged accordingly and requests created that would reproduce said data changes.

The obvious downside to this approach is its complexity and the need to adhere closely to the server architecture. Any changes on the backend must be properly mirrored on the frontend and the thickness of the client therefore grows, meaning more expensive development and maintenance. On the other hand, the robustness of such solution is

16. https://developer.android.com/training/data-storage/room
17. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

incomparable to the previous one, because there is strong consistency in the data (all relevant data is always updated properly, which may not be true in the first approach if there are complex dependencies across entitites).

2.2.3 Hybrid solution

There is also a "hybrid solution" — one that uses the database which is controlled by the middleware. Effectively, the application is completely unbothered by being offline, but the requests sent to the server are intercepted by the middleware, processed, translated to local database queries, and also stored to a pending operations queue. The GET requests are transformed to retrieve data from the local database, because all changes are reflected in it.

Upon getting back online, the synchronization is as simple as in the first approach — sending the requests to the server in the order of interception. Combining the strengths of both approaches, the cost is in the increased complexity.

Now, effectively, the backend together with the database is completely mocked in the client. This significantly increases costs and maintainability of such a solution and therefore should be used in specific scenarios only, especially when there are complex data dependencies and high independence requirements (the client must remain offline for longer periods of time).

2.3 Synchronization

Regardless of the choice of the local data handling and therefore synchronization method, there may always arise conflicts. Conflicts arise when the offline and online data are both changed and therefore one change may overwrite the other.

To illustrate the approaches described in this section on a particular scenario, let's assume this situation: there are two users, Alice and Bob. They both work in the same time with the same system and they interact with the same data, or at least a portion of the data they work

with is shared. Alice works online, while Bob is in offline mode. After some time, Bob reconnects to the internet and tries to synchronize his changes, creating conflicts on the shared data.

2.3.1 Acceptance of conflicts

The simplest way of handling conflicts is to simply not handle them at all and accept their existence. Therefore, Bob, after succesfully connecting to the internet after his offline session, overwrites any data that Alice had submitted to the same instances that Bob worked with. The whole instance will be always overwritten.

In case there is a stateful instance and Alice has changed its state to one that does not accept changes, Bob's request synchronization will fail. If the server stores the history of states of the instance with timestamps and the client sends a timestamped synchronization request (so the request will be processed as if it was received when Bob performed the update and not in the moment of reception), Bob's request may succed — or it still may not if the timing is unfortunate.

There are also other cases like when Alice deletes the instance and Bob tries to update it — which fails. When there is a state machine on the instance, it may reject Bob's state updates as they are invalid at the moment of request processing (or even in the moment of Bob's action when timestamping is involved) — because Alice changed the state as well. In the simple case of *New* – *Processing* – *Processed* state machine, if both Alice and Bob change the state to *Processing*, the latter must fail as there is, presumably, no path from the *Processing* state to itself.

2.3.2 Avoidance by locking

One of the most robust yet the most restrictive approach is for Bob to lock preemptively any instances that he intends to interact with while offline, effectively completely avoiding the risk of any conflict.

An instance-granular locking mechanism must be implemented, and Bob, before going offline, has to be able to mark the instances that he needs in offline and lock them. Optionally, instances might be

locked together with all their "dependent (child) instances" — that is, instances that have a $n : 1$ relation with said instance. This approach solves all the update-delete and stateful problems.

There are, unfortunately, several problems to this approach — firstly, Alice is blocked from working with the instances completely. Even though they might not overwrite each other's attributes as the attribute subsets do not overlap, she shall not interact with the instances at all. Secondly, Bob has to know what data he will interact with in advance.

And thirdly, after he is done working with the data, he must always unlock the instances on the server — failure to do so will block any other user from interacting with it over potentially a prolonged period of time. Assuming Bob went on holiday for a week and did not bring his work device with him, the only option for his colleagues to work with the data is to have a system administrator that can forcefully unlock the instances, effectively scrapping Bob's unsaved changes.

2.3.3 Mitigation by partitioning

The risk of synchronization conflicts can also be somewhat mitigated by partitioning the data into several parts, each having their own mutating endpoint. Let's assume that Alice sits in an office and Bob is operating outside and they cooperate on a shared instance representing a physical object. Bob has physical access to the object. It is probable that some attributes can only be filled in terrain while others should be filled in the office.

When the instance has this conceptual split of attributes and it makes sense in the particular context, the update endpoint can be split in two — the office and terrain one, minimizing conflicts.

If the backend endpoint structure supports PATCH requests to update only selected attributes and not overwrite others in the process, it is the same idea but with attribute granularity. While this partially mitigates the risk, it certainly does not handle it completely and also does not solve the update-delete or stateful cases.

2.3.4 Avoidance by instance duplication

There is an option to duplicate the instance if a conflict during the synchronization would have to occur[18]. The simplest implementation would be to set up a versioning system where each instance stores its version (as a numerical attribute). When processing an update request, the server checks the incoming version and compares it against the stored one. If they match, the update is performed. If not (e. g., server has version 35 and an update request with version 34 is received), the server duplicates the instance and flags it as a duplicate.

The server may also resort to creating a duplicate when a deleted instance is attempted to be updated (of course, provided that soft-deleting is employed and the instance still exists in the database) or when a state machine restrictions would prohibit the request on a stateful instance to be performed.

The responsibility is then on the system administrator or whoever might be identified as the instance owner to resolve the duplicity later and merge the duplicates manually.

There is then the question about any dependent instances[19] — should they be duplicated as well, performing a deep duplication, or only a shallow one, duplicating the dependent instances on demand? The answer relies on the data model. If there are only a few dependent entities, it will probably be easier to perform a deep duplication. But in case of a complex data model, the approach does not scale well and a more complicated on-demand duplication scheme should be applied. That means that the dependent entities are not duplicated, but their versions are incremented, ensuring that any further synchronization requests that would reference these dependent instances would produce a version conflict, triggering a duplication.

---

18. The synchronization requests must be flagged as duplication-enforcing — normally, when the version does not match during online usage, the server most likely just returns a version error, forcing the client to refetch the current version and performing their changes again. That, however, cannot be done when synchronizing changes from an offline session.
19. Dependent instances is a concept introduced in section 2.3.2 — instances of entities that have a $n : 1$ relation with the entity at hand and may therefore be perceived as subordinate.

There is one last remark to be made about this approach: the server must notify the client about any duplicates being created and the ID mapping of originals and duplicates. The client then must respect this mapping in all following requests to affected instances.

2.3.5 Mitigation by atribute timestamping

In some cases, the best option may be to timestamp the last update of every attribute of an entity. While effectively doubling the attribute count, it is guaranteed that only the latest update of every attribute is persisted. For the backend, it means updating per attribute while always checking which value is the latest. For the frontend, it usually means managing the local data in a mirrored local database that also reflects this procedure, and also implementing an attribute-precise timestamp control in any forms in the UI.

This method of conflict resolution is often referenced as "Last Write Wins" and there are database systems specifically designed to utilize this strategy like Apache Cassandra[19], a distributed NoSQL database that timestamps every attribute.

2.3.6 Advanced synchronization conflicts resolution concepts

There are countless ways to design a custom solution finetuned to specific business requirements. The following concepts can also be employed when creating a more specialized approach:

CRDT

A CRDT (Conflict-free Replicated Data Type) is a data structure that ensures eventual consistency between multiple nodes of a distributed peer-to-peer system without a central node, even when updates occur concurrently.

As Shapiro et al. (2011)[20] explain, each node can independently modify its local copy and propagate the changes through the network, and the CRDT's merge function guarantees that all nodes eventually

converge to the same final state once all updates are propagated. It is however essetial to be able to create a merge strategy for each attribute. A merge strategy defines how conflicting updates are resolved in a deterministic way. It strongly depends on the attribute type: sometimes it is taking the latest timestamp, sometimes merging sets, sometimes combining counters, etc.

The context od this thesis is somewhat related — there is a central node (the server), but each client may have their truth and there needs to be a strategy to determine what is the final shared truth per attribute.

In practical use, this can be represented by a key-value structure where each client (key) sends its own version of a value, optionally tagged with timestamp or version. The server holds all the key-value pairs and figures out a final value for each attribute that is then propagated to all clients.

In reality, there may be data types that cannot have a simple deterministic strategy to merge multiple versions. In that case, the UI can display all the key-value pairs for an attribute to the instance owner and let them manually merge the conflict and choose a single value that will be stored instead of all the conflicting items.

GIT

Talking about resolving merge conflicts should ring a developer's bell — there is a versioning system that handles merging data from multiple users and entails resolving merge conflicts: GIT. Its core concepts may be transformed to tackle issues in the context of this thesis.

An object-oriented *diff* alternative might be implemented to determine potentially diverging attributes together with a custom merge editor for merging those diversions that cannot be merged automatically. In GIT terminology, attributes would mirror lines, sending a PUT request is pushing a commit and the server performs a fast-forwarding or merging when processing the update.

2.3.7 Final synchronization remark

There is one remark that has yet to be made concerning the aforementioned approaches. Neither of them is globally better or worse than any other. Choosing the right one depends heavily on the functional and non-functional requirements of a particular project and its business logic and context. The first, and the most important step, is always to consult the business aspects of the offline functionality with the customer and depending on the specific requirements one can then choose an adequately robust and complex solution.

3 Case study

To illustrate the real designing process of an offline-enabled application and all the architectural decisions based on the guide from chapter two, in this chapter will be documented a case study of a real-world solution.

3.1 Functional and non-functional requirements

3.2 Architecture, technology stack

This application needs to be supported both on mobile devices with Android or iOS and on Windows devices. As in the majority of enterprise applications, there is no need for resource-intensive calculations, no performance-critical tasks and there is minimal operating system integration — the only functional requirement in this area that needs to be fulfilled is camera support.

Therefore, the only options left are React PWA and Flutter. In this case, the browser-first PWA option won as it is easier to integrate into a web ecosystem, can be consistently used on any device equipped with a browser, does not have to be installed, and quite importantly, the updates are fully managed by the application. That means eliminating any version inconsistencies and therefore potential issues with a user base that tends to be reluctant to update applications unless it is forced to.

As the programming language, TypeScript[1] was chosen for type safety to help minimize runtime errors. TypeScript alows using static typing, generics or interfaces. Vite[2] was chosen as the build tool for its speed and features. On top of the React-TS PWA platform, a reliable and powerful set of libraries was chosen to ensure stability and extensibility. To mention a few of the most prominent:

1. https://www.typescriptlang.org/
2. https://vite.dev/

1. Tanstack Query[3] is a library for request, state, and asynchronous data management — primarily used for consistent, state-aware handling of fetching and mutating any data.

2. Axios HTTP[4] is a promise-based HTTP client with concise syntax and powerful tools like interceptors or proxies.

3. React Hook Form[5] is a modern, fully featured, UX oriented, and flexible library to be used for form handling in React apps with state management and broad validation support.

4. Zod[6] is a TypeScript-first validation library that enabled building complex validation schemas with the option to infer types from the schemas to use. It is fully compatible with React Hook Form.

5. Tailwind CSS[7] is a concise and expressive CSS framework that is utility based. It minimizes CSS bundle size while providing a fast and powerful way to build UIs.

6. ShadCN[8] is a highly customizable and extensible component library that is based on Tailwind CSS.

3.3 Service Worker

What a service worker is has been explained in detail in chapter two. Here the focus is on the actual implementation with emphasis on the different strategies to be employed in different scenarios.

3. https://tanstack.com/query/latest
4. https://axios-http.com/
5. https://react-hook-form.com/
6. https://zod.dev/
7. https://tailwindcss.com/
8. https://ui.shadcn.com/

3.3.1 Workbox + Vite

Workbox[9] is a high-level Service Worker library developed by Google to simplify working with service workers. It provides a set of modules to abstract over the complex service worker API.

Inside of Vite config, there is a manifest for the PWA to be injected into the build. Inside it, things like application name or icon are specified. There is also the reference to the service worker source code file. This file, `prompt-sw.ts` includes these logical parts: the setup and the caching strategies

Listing 3.1: Setup

```
import {
  cleanupOutdatedCaches,
  createHandlerBoundToURL,
  precacheAndRoute,
} from "workbox-precaching";
import { NavigationRoute registerRoute } from "
   workbox-routing";

declare let self: ServiceWorkerGlobalScope;

// Activate SW immediately upon installation -
   apply updates instantly
self.addEventListener("message", (event) => {
  if (event.data && event.data.type === "
     SKIP_WAITING") self.skipWaiting();
});

// self.__WB_MANIFEST is default injection point
precacheAndRoute(self.__WB_MANIFEST);

// Clean old assets
cleanupOutdatedCaches();

// To allow work and navigation offline
```

9. https://developer.chrome.com/docs/workbox

```
registerRoute(new NavigationRoute(
   createHandlerBoundToURL("index.html")));
```

Listing 3.2: Caching

```
import { Route, registerRoute } from "workbox-
   routing";
import {
  CacheFirst,
  NetworkFirst,
  StaleWhileRevalidate,
} from "workbox-strategies";

// Handle backend requests
const backendRoute = new Route(
  // Match API requests
  ({ url }) => url.pathname.startsWith("/api"),
  // Use NetworkFirst strategy to fetch data
     from the network first
  new NetworkFirst({
    cacheName: "api",
  }),
);

// Handle images:
const imageRoute = new Route(
  ({ request }) => {
    return request.destination === "image";
  },
  new StaleWhileRevalidate({
    cacheName: "images",
  }),
);

// Handle scripts:
const scriptsRoute = new Route(
  ({ request }) => {
    return request.destination === "script";
```

```
  },
  new CacheFirst({
    cacheName: "scripts",
  }),
);

// Handle styles:
const stylesRoute = new Route(
  ({ request }) => {
    return request.destination === "style";
  },
  new CacheFirst({
    cacheName: "styles",
  }),
);

registerRoute(backendRoute);
registerRoute(imageRoute);
registerRoute(scriptsRoute);
registerRoute(stylesRoute);
```

3.3.2 Workbox strategies

As can be seen in the provided example, there are different caches and caching strategies used for different offline assets. For API requests, the Network-First strategy is used. That means, whenever there is access to the network and a request is intercepted by the service worker, fresh data is always fetched and if applicable, stored in the `api` cache[21]. Only when the server is not reachable, cached data is served. This ensures that whenever possible, the latest data is fetched and presented.

Second strategy used, Cache-First, is utilized for styles and scripts — in other words, parts of the source code of the application itself. The strategy, as the name suggests, fetches the data straight from the cache and requests the server only if the data is not present in the cache[21]. This approach is used because on every applictation update and therefore service worker registration, the cache is cleaned by

the `cleanupOutdatedCaches` function. Therefore, the aforementioned resources are cached at the longest until a new application version arrives, which is perfectly fitting for this type of assets — the cache is, technically, never outdated and loading the data from it is way quicker than fetching it from the server.

The last of the used strategies is called Stale-While-Revalidate. It fetches the data from the cache if possible and returns them to the client, while fetching a fresh version from the server and upon receiving a response, it updates the cache[21]. The data might not be fresh everytime, but they are *eventually consistent*, while prioritizing load times. The strategy is used for images, as those may usually take a while to load and their freshness is not critical.

3.3.3 Vite PWA

The Vite PWA plugin[10] builds upon the Workbox to provide ready to use React hooks like `useRegisterSW` that handles registering and updating the application. In this case, as has been foreshadowed earlier, the updates are forced upon the user, as illustrated by the `useEffect` hook used for updating the service worker as soon as an update is detected. The following snippet is simplified to convey the main point, its full content is attached as an electronic attachment in the file `reload-prompt.tsx`.

Listing 3.3: Service worker registration and updates management

```
import { useRegisterSW } from "virtual:pwa-
   register/react";

export function ReloadPrompt() {
  const { offlineReady, needsRefresh,
     updateServiceWorker } =   useRegisterSW({
    onRegisteredSW(swUrl, r) {
      console.log(`Service Worker at: ${swUrl}`)
         ;
      setInterval(
```

10. https://vite-pwa-org.netlify.app/

```
        () => {
          console.log("Checking␣for␣sw␣update...
             ");
          r.update();
        },
        5 * 60 * 1000, // 5 min
      );
    },
    onRegisterError(error) {
      console.error("SW␣registration␣error",
         error);
    },
  });

  useEffect(() => {
    if (needRefresh) {
      updateServiceWorker(true);
    }
  }, [needRefresh]);


  return (
    <>
      {offlineReady && (
        <>
          Offline mode is ready
          <Button onClick={close}>Close</Button>
        </>
      )}
      {needsRefresh && (
        <>
          A new app version available
          <LoadingSpinner />
          Updating...
        <>
      )}
    </>
```

```
  );
}
```

3.4 Local data management

In this application, the majority of data is not created, but mutated. Prepared Orders are fulfilled, activities completed, data filled, there is only a few usecases that create data, like, for example, attachments. The POST-PUT chaining problem therefore might be ommited as updating created instances may be forbidden, while only allowing deleting the instance and re-creating it if necessary. A lot of the data is also state-dependent and timing and order of operations crucial — the Order state dictates what data can be accesed and generally what can be done in the app.

For those reasons, the intercepting approach was chosen for managing local data. Therefore there is no local mirrored database, only a middleware that intercepts requests in offline mode and stores them in a queue. That FIFO[11] queue is then queried for any staged data and synchronized after reconnecting to the internet.

To answer the question why the hybrid approach was not chosed as it is more robust, the main point is that the requirements towards considerable offline independence are not present. The application will not remain offline for more than a work shift and the typical scenario is spending only a couple of hours offline and then coming back online. The application is not request-intensive and managing a mirrored local database under the middleware would, in this case, be excessive.

3.4.1 Axios Interceptor

The intercepting middleware is implemented as an Axios interceptor. Axios, being the HTTP client, offers an option to set up two interceptors, one for requests and one for responses. Each interceptor has

11. First In, First Out

a processing function that may modify the payload, react to it in a custom way or even capture it and not propagate further[22]. For response interceptor, it may mean that every response with an error code is displayed as an error snackbar in one place intead of handling it per request. For request interceptor, it may be used for capturing offline requests.

The request interceptor used in this application is set up in the following way[12]:

1. It queries the browser if it is online.
2. If so, it just passes the request.
3. If not, it enqueues the request into the request FIFO and aborts the original request. For this purpose, in the response interceptor, there is a filter for aborted responses — they are not to be propagated into the app as they have been aborted by the request interceptor.

The request FIFO itself is persisted in the browser's localStorage in order to survive app reloads[13]. It stores the so-called *Axios configs* — objects storing all the relevant attributes, data, metadata, and options for a request, like url, method, request body, parameters, headers, flags, transformers or proxies[23]. Later, when online, the config can just be passed to Axios, which composes the request and transmits it.

3.4.2 Incorporating the staged data

When offline, the application fetches data as normal (they are, in fact, served from cache by the service worker), then retrieves staged data from the FIFO, and finally decides how to merge the fetched data with the staged data — overlaying them, proritizing one over another or any other merging strategy. As an example, let's take the Activity detail. The Activity is fetched from `/api/activity/<activityId>`, its staged data retrieved by `getStagedActivityData` from the queue, and the

12. The relevant code is in `src/api/axios-config.ts`.
13. Its code is in `src/offline/request-fifo.ts`

component that renders it, in this case, prioritizes the staged data over the fetched data.

Listing 3.4: Staged data retrieval

```
  export function getStagedActivityData(
  activityId: string,
): ActivityPerformedData | undefined {
  const fifo = getFifo();
  // go from start of array - last updates
  for (const req of fifo) {
    if (
      req.request.url?.includes("activity") &&
      req.request.url?.includes(activityId)
    ) {
      return req.request.data;
    }
  }
  return undefined;
}
```

Listing 3.5: Simplified data enrichment

```
  const methods = useForm<
     ActivityPerformedDataForm>({
    resolver: zodResolver(
       activityPerformedDataFormSchema),
    defaultValues:
      stagedActivityData
        ?? freshActivityData
        ?? defaultActivityData
  });
```

3.5 Synchronization

The synchronization approach in this aplication is a combination of mitigation and acceptance: it is expected that each user has their own data that they work with. In case there might be two users cooperating

on the same order, they each have their part of data that they interact with. In the unexpected (and technically unsupported) case they do not cooperate, the risk is simply accepted: they will overwrite each other's data (last write wins) or they might cause conflicts when the pending changes are no longer applicable to the current version of the data.

The stored Axios configs (requests) can be seen in the application as pending in the synchronization dialog. They have a timestamp when they have been recorded, which is then sent as *validFrom* query parameter to the backend during synchronization. Each config is resent by Axios in the order in which they have been recorded. There are some requests that can be aborted (and, therefore, deleted from the queue) — yet not all of them should have the option to be aborted, for example, all state-dependent requests and state-changing requests should not be tampered with as they can corrupt the validity of the following requests.

If a request fails during the synchronization, the user has an option to either try again, delete the failing request if possible, or throw away all the offline changes.

TODO screenshoty

4 Selected issues

4.1 Offline testing

4.2 Dependent requests

file uploads -> separate queue

4.3 ?

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.microsoft.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.

15. REACT TEAM. *Versions – React* [online]. Meta Platforms, Inc., 2025. [visited on 2025-10-27]. Available from: https://react.dev/versions.

16. REACT TEAM. *Writing Markup with JSX* [online]. Meta Platforms, Inc., 2025. [visited on 2025-10-27]. Available from: https://react.dev/learn/writing-markup-with-jsx.

17. LEPAGE, Pete; RICHARD, Sam. *What are Progressive Web Apps?* [online]. Google LLC, 2020. [visited on 2025-10-27]. Available from: https://web.dev/articles/what-are-pwas.

18. FIRTMAN, Maximiliano. *Service workers* [online]. Google LLC, 2021. [visited on 2025-10-27]. Available from: https://web.dev/learn/pwa/service-workers.

19. YARABARLA, Sandeep. *Learning Apache Cassandra – Second Edition*. Packt Publishing, 2017. ISBN 9781787127296. Available also from: https://www.oreilly.com/library/view/learning-apache-cassandra/9781787127296/.

20. SHAPIRO, Marc; PREGUIÇA, Nuno; BAQUERO, Carlos; ZAWIRSKI, Marek. Conflict-Free Replicated Data Types. In: DÉFAGO, Xavier; PETIT, Franck; VILLAIN, Vincent (eds.). *Stabilization, Safety, and Security of Distributed Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN 978-3-642-24550-3.

21. CHROME DEVELOPERS. *Strategies for service worker caching – Workbox* [online]. Google LLC, 2021. [visited on 2025-10-27]. Available from: https://developer.chrome.com/docs/workbox/caching-strategies-overview.

22. AXIOS DOCUMENTATION TEAM. *Interceptors · Axios Docs*. 2025. Available also from: https://axios-http.com/docs/interceptors. accessed: 2025-11-14.

23. AXIOS DOCUMENTATION TEAM. *Request Config · Axios Docs*. 2025. Available also from: https://axios-http.com/docs/req_config. accessed: 2025-11-14.

A Electronic attachments