BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

COLLABORATIVE DATA SHARING IN A TIME-MANAGEMENT APPLICATION

KOLABORATIVNÍ SDÍLENÍ DAT V APLIKACI PRO SPRÁVU ČASU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAN ZIMOLA**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ HYNEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

165062

Institut:	Department of Information Systems (DIFS)
Student:	**Zimola Jan, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Application Development
Title:	**Kolaborativní sdílení dat v aplikaci pro správu času**
Category:	Mobile applications
Academic year:	2024/25

Assignment:

1. Study task planning and synchronization across devices and users, focusing on relevant technologies and tools.
2. Investigate smartwatches, including their hardware, platforms, and development technologies.
3. Analyze the current state of the TimeNoder2 application, focusing on architecture and user feedback. Evaluate the existing support of synchronization between devices and users. Take into account the absence of support for smartwatches.
4. Based on the analysis, extend the existing TimeNoder2 mobile application to support synchronization between devices and users, and design a variant of the smartwatch application.
5. Implement the proposed TimeNoder2 extensions.
6. Test the application in a real environment and propose possible extensions.

Literature:

- Google. (n.d.). *Android developers*. Android Developers. Retrieved September 30, 2024, from https://developer.android.com
- Google. (n.d.). *Flutter documentation*. Flutter. Retrieved September 30, 2024, from https://docs.flutter.dev
- Knott, D. (2015). Hands-On Mobile App Testing: A Guide for Mobile Testers and Anyone Involved in the Mobile App Business. Pearson Education.
- Krug, S. (2013). Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability. Pearson Education.
- Liu, R., & Lin, F. X. (2016, June). Understanding the Characteristics of Android Wear OS. In *Proceedings of the 14th annual international conference on mobile systems, applications, and services* (pp. 151-164).

Requirements for the semestral defence:
Items 1 - 4.

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

Supervisor:	**Hynek Jiří, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	22.10.2024

Abstract

This thesis lays the groundwork for future cross-platform support and user collaboration in TimeNoder2. It introduces two synchronization strategies: one for a single user across multiple devices, based on the WatermelonDB protocol, and another for multi-user collaboration across devices, utilizing the existing PowerSync synchronization framework. A key contribution is the implementation of shared documents with conflict resolution and full offline capability, built upon the AppFlowy editor. The project also extends TimeNoder2 to WearOS using Flutter, resulting in a standalone smartwatch application with persistent offline functionality. This design addresses several limitations found in existing smartwatch apps, such as TickTick and Focus ToDo.

Abstrakt

Tato práce pokládá základy pro budoucí podporu různých platforem a spolupráci uživatelů v TimeNoderu2. Zavádí dvě synchronizační strategie: jednu pro jednoho uživatele na více zařízeních, založenou na protokolu WatermelonDB, a druhou pro spolupráci více uživatelů na různých zařízeních, využívající stávající synchronizační rámec PowerSync. Klíčovým přínosem je implementace sdílených dokumentů s řešením konfliktů a plnou možností práce v režimu offline, postavená na editoru AppFlowy. Projekt také rozšiřuje TimeNoder2 o podporu operačního systému WearOS pomocí nástroje Flutter, což vede k vytvoření samostatné aplikace pro chytré hodinky s trvalou offline funkcí. Tento projekt řeší několik omezení, která se vyskytují ve stávajících aplikacích pro chytré hodinky, jako jsou TickTick a Focus ToDo.

Keywords

Synchronization protocols, conflict-free replicated data types, collaborative editing, smartwatch applications, WearOS development, Flutter framework, offline-first design, PowerSync, WatermelonDB protocol

Klíčová slova

Synchronizační protokoly, bezkonfliktní replikované datové typy, společné úpravy, aplikace pro chytré hodinky, vývoj WearOS, framework Flutter, offline-first design, PowerSync, protokol WatermelonDB

Reference

ZIMOLA, Jan. *Collaborative Data Sharing in a Time-Management Application.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Hynek, Ph.D.

Collaborative Data Sharing in a Time-Management Application

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Jiří Hynek, Ph.D. I have cited all literary sources, publications, and other resources from which I drew information. I declare that I used GitHub Copilot and ChatGPT to assist with code generation, debugging, and exploring implementation ideas, and GrammarlyAI and Writefull to improve the clarity and phrasing of the written text. All final decisions regarding content, structure, and implementation were made by me.

.....
Jan Zimola
May 21, 2025

Acknowledgements

I would like to express my heartfelt gratitude to my family for their unwavering support in my pursuit of higher education. Their encouragement has allowed me the freedom to explore my passions and develop my skills. I am also deeply thankful to my supervisor, Mr. Ing. Jiří Hynek, Ph.D., for his remarkable patience and prompt assistance throughout this journey.

Contents

List of Figures

Chapter 1

Introduction

In today's digital age, technology is deeply integrated into our daily lives, offering tools to manage increasingly complex schedules and demands. Effective time management is critical, and mobile apps have become essential for organizing tasks, setting deadlines, and tracking progress directly from our devices. Among these tools, TimeNoder2[1] has gained attention but faces growing user demand for seamless cross-platform support and collaborative features. By cross-platform support is meant both being able to open the application on multiple platforms, but also to share data between them. At the same time, it is important to allow work on these platforms offline, which clearly indicates that these devices need to synchronize.

This work aims to prepare TimeNoder2 to become a fully cross-platform application. It looks at important challenges such as data synchronization and collaborative features. The goal is to ensure that user data, organized like a database, remains consistent across all devices. Additionally, the thesis explores how users can share content in text editors for real-time collaboration, even offline. Another focus is on extending TimeNoder2 to WearOS[2] using Flutter[3], allowing the app to work independently from the mobile version. This could solve issues seen in other apps like Focus Todo[4] and TickTick[5]. Instead of coming up with completely new ideas, this thesis adapts existing synchronization methods and tools to fit TimeNoder2's needs. It uses known protocols to enhance the app's functions and better serve users. The result is a strong foundation for TimeNoder2 to become a flexible, cross-platform tool that helps users stay organized across their devices.

The structure of the thesis is as follows: Chapter 2 discusses data synchronization concepts and tools. Chapter 3 looks at smartwatches and how they fit into a cross-platform system. Chapter 4 evaluates the current status of TimeNoder2 and user expectations based on reviews and comments, focusing on cross-platform support. Chapter 5 describes the design of the synchronization system and smartwatch application. Chapter 6 covers synchronization implementation using tools like WatermelonDB[6] protocol, PowerSync[7], and CRDTs, along with the smartwatch app. Finally, Chapter 7 explains the testing process to ensure synchronization and the smartwatch app's reliability and performance.

[1] https://timenoder.com/
[2] https://wearos.google.com/
[3] https://flutter.dev/
[4] https://www.focustodo.cn/
[5] https://ticktick.com/
[6] https://watermelondb.dev/docs/Sync/Backend
[7] https://www.powersync.com/

Chapter 2

Task scheduling and synchronization

Accessing the application from multiple devices is essential, as seen in other time-management applications like TickTick and Todoist[1] [46]. This includes web, mobile, and smartwatches. Cross-platform functionality allows for convenient usage on desktop PCs, which provide significantly more screen space. It also enables basic tasks to be performed on smartwatches. Another vital aspect is collaborating on tasks with others. This can apply to school projects, family organizations, or work-related activities. These methods are well integrated into TickTick and Todoist based on PCMag[2]. The cross-platform support relies on the synchronization of data across devices.

Data synchronization is the technique to ensure that data across multiple locations remains consistent and harmonized as it changes. This practice is applied in various applications and involves specific algorithms designed for syncing files. In computer science, data synchronization refers to the maintenance of data integrity and the coherence of multiple copies of a dataset with each other [24].

When devices are offline, they record changes in their local database. Upon reconnecting, they send these changes to the server, synchronizing them to the database [24].

2.1 Distributed Systems

Mobile applications that share data across multiple devices and server databases are examples of distributed systems [66, 57]. There are several definitions for distributed systems. One such definition states: *"A distributed system is a collection of autonomous computing elements that presents itself to users as a single, coherent system [57]."*

A distributed system consists of independent computing elements, called nodes, which can be hardware devices or software processes. Users perceive these nodes as a single system, requiring collaboration among them. Establishing this collaboration is essential for developing distributed systems. There are no assumptions about the types of nodes, which can range from mainframe computers to small sensor devices [57].

Nodes operate independently while working towards common goals by exchanging messages. Each node has its perception of time, as there is no global clock, which creates challenges for synchronization and coordination in distributed systems. However, it is not

[1] https://todoist.com/
[2] https://www.pcmag.com/reviews/

always necessary to work with real-time. What is usually only needed is the ordering of events. This is why vector clocks or hyper logical clock (HLC [43]) are used [57].

A significant aspect of distributed systems involves laptops and mobile devices with power and connectivity limitations. Companies that manufacture these devices are working on ways to extend their operational time without requiring large batteries, which can impede usability. Power efficiency can be achieved through effective software as well as efficient hardware [67, 33].

To conserve energy, the device may temporarily terminate certain processes or disable the network . Managing group membership within a distributed system is a complex task. In traditional distributed systems, a loss of connection is typically viewed as a failure. However, the necessity to conserve energy introduces additional challenges, as we need to distinguish between planned, controlled disconnections and those that are forced. Furthermore, each communication session incurs a high start-up cost [66].

2.2 Eventual Consistency

Distributed systems are limited by the so-called CAP theorem, which states that it is impossible to guarantee consistency, availability, and partial tolerance at the same time. It is possible only to have two at the same time [48]:

- Consistency (C) – *"All nodes share the same states, that is, they all have the same data. In an informal manner, a system is consistent if a write is successful, all the components of the system can read the new value."*
- Availability (A) – *"The system remains operative to take care of every client request, managing it and answering it. Furthermore, it also means that the system is still awake even if a node fails (or crashes)."*
- Partial Tolerance (P) – *"The system keeps attending client requests even though it has been divided into at least two different parts, also called partitions, that cannot communicate with each other."*

In the case of time-management applications, we require high availability. This means that we want users to be able to create tasks and perform other actions even while offline. As a result, we need to sacrifice some consistency. This approach is known as the "optimistic style", which allows operations to be executed on the partition that holds the data without limiting availability. However, this method may lead to global inconsistencies [22]. This approach is known as eventual consistency, in which changes made to one copy will eventually propagate to all copies. If the update activity ceases, all copies will converge to the same state after a certain period of time [35, 61].

Eventual consistency requires receiving the events in the same order on all nodes. This is hard to guarantee. This is why special structures called Conflict-free replicated data types (CRDTs) emerged that guarantee Strong Eventual Consistency. An object is considered Strongly Eventually Consistent if it is both Eventually Consistent and Strongly Convergent, meaning that correct replicas that have received the same updates will have equivalent states [56].

Another approach is Operation Transformation (OT), which is used in applications like Google Docs. However, OT is seen as outdated because many of its algorithms have been found to have errors, and the ones that work need a centralized server. This dependency is

a major limitation, leading the community to focus on Conflict-free Replicated Data Types (CRDTs) instead. As a result, there are many libraries available for CRDTs. Therefore, this thesis will focus solely on CRDTs [41].

2.3 Conflict-free replicated data types (CRDTs)

In the eventual consistency model, replicas of data are allowed to diverge temporarily but are expected to converge to the same value if no further changes occur. One way to achieve eventual consistency is through the use of Conflict-free Replicated Data Types (CRDTs) [56]. CRDTs can be classified into two categories: *state-based* and *operation-based* [47].

2.3.1 State-based CRDTs

The *State-based Convergent Replicated Data Type* (**CvRDT**) provides reliable convergence in distributed systems by ensuring that all replicas ultimately agree on a single unified state, regardless of the order or timing of updates. This is accomplished through the use of a structure known as a *join-semilattice* [56].

Join-Semilattice

A *join-semilattice* is a mathematical structure that guarantees the existence of a way to combine every pair of states into a single state using a **Least Upper Bound (LUB)** operation. This LUB operation has several key properties [56]:

- **Commutativity:** $x \cup y = y \cup x$,
- **Idempotence:** $x \cup x = x$,
- **Associativity:** $(x \cup y) \cup z = x \cup (y \cup z)$.

Monotonic Semilattice

A Monotonic Semilattice is an extension of *Join-Semilattice* that includes additional properties [56].

1. **Merging:** The merge operation combines two states s and s' by computing their LUB: $s \cdot m(s') = s \cup s'$.
2. **Monotonic Updates:** States grow monotonically over time, meaning that updates only add information and do not remove or modify it: $s \leq s \cdot u$.

Example: Distributed To-Do List

To illustrate the concept of a CvRDT, consider a *distributed to-do list* shared across multiple devices. Each device maintains its own copy of the list, and the goal is to ensure all devices converge to the same state after synchronizing.

State Representation Each device's to-do list is represented as a **set of tasks**. For example, one device may have the state $S_1 = \{\text{Task A, Task B}\}$, while another has $S_2 = \{\text{Task B, Task C}\}$.

Merge Operation When two devices synchronize, their states are merged using the **union** operation, which is the LUB for sets:

$$S_1 \cup S_2 = \{\text{Task A, Task B, Task C}\}.$$

The union operation satisfies:

- **Commutativity:** The order of merging does not matter.
- **Idempotence:** Merging the same state twice has no effect.
- **Associativity:** Merging multiple states in different orders gives the same result.

Monotonic Updates The updates to the to-do list are monotonic, which means that while new tasks can be added, existing tasks are neither removed nor changed.

Convergence Using the merge operation ensures that all replicas eventually converge to the same state. If one device adds *Task D* while another adds *Task E*, their states will merge into:

$$S = \{\text{Task A, Task B, Task C, Task D, Task E}\}.$$

2.3.2 Operation-Based Commutative Replicated Data Type (CmRDT)

An *Operation-based Replicated Data Type* (**CmRDT**) describes distributed objects where updates are explicitly sent and applied at all replicas. Unlike state-based CRDTs, CmRDTs split an update into two parts [56]:

1. **Prepare-update** (t): A side-effect-free operation performed locally at the source replica.
2. **Effect-update** (u): The actual operation broadcasted to all replicas and applied downstream.

Causality

In the following definitions, the definition of causality will be used. Causality determines the sequence of events, with a cause always preceding its effect. In distributed systems, this principle is evident as a chain of actions: one node reads data and writes a new value, which another node later uses for its operation. These interconnected actions establish the causal order, defining which events occurred before others [40].

Principle

- **At the source replica**, when an operation is invoked, the prepare-update (t) is executed to determine the effect-update (u). For example, when a replica performs an increment operation on a counter, (t) determines how the counter's state changes, while (u) defines the specific change, such as incrementing the counter by 1.
- **At downstream replicas** the effect update (u) is communicated to all replicas using a reliable communication protocol, such as *causal broadcast*. Each replica applies (u) in the same order to ensure consistency.

Causal History

The *causal history* tracks the sequence of operations at each replica to ensure causality between updates. It ensures that prepare-updates (t) and effect-updates (u) are causally ordered, such that u only applies if all prior updates in the causal chain have been delivered.

Commutativity

A CmRDT requires that all concurrent operations satisfy *commutativity*. That is:

- For any two updates (t, u) and (t', u'), applying them in any order results in the same state. This ensures that replicas converge even when operations are delivered in different orders.
- If operations are not *commutative*, they need to be *causally ordered*.

Example: CmRDT Counter

Consider a distributed counter replicated across devices. Let t be the prepare-update that increments the counter locally, and `u = incrementByOne` be the effect-update broadcasted to all replicas. When a user increments the counter:

1. **At the source replica**:
   - t updates the local counter.
   - `u = incremenetByOne` is generated and sent to other replicas.
2. **At downstream replicas**:
   - u is applied, ensuring the counter increases by 1.

Because `incrementByOne` can be applied in any order, all replicas converge to the same counter value.

2.3.3 CRDT Set Types With Examples

Concrete examples of CRDT types can be implemented using either *state-based* or *operation-based* approaches [56]:

- G-Set (Grow-Only Set),
- 2P-Set (Two-Phase Set),
- LWW-Set (Last Writer Wins Set),
- PN-Set (Positive-Negative Set),
- OR-Set (Observed-Remove Set).

1. G-Set (Grow-Only Set)

A G-Set allows only `additions` of elements. Once an element is added, it cannot be removed [56, 47].

Example:

- At $Replica_1$: $add(a)$, $add(b)$
- At $Replica_2$: $add(c)$

The states at the replicas are:

$$\text{State of } Replica_1 = \{a, b\},$$

$$\text{State of } Replica_2 = \{c\}.$$

When merged:

$$\text{Merged State} = \{a, b, c\}.$$

2. 2P-Set (Two-Phase Set)

A 2P-Set allows elements to be `added` and `removed`, but once removed, an element can never be added again. It uses two sets [56, 47]:

- A: Tracks added elements.
- R: Tracks removed elements.

Example:

- At $Replica_1$: $add(a)$, $remove(a)$
- At $Replica_2$: $add(a)$, $add(b)$

The states at the replicas are:

$$\text{State of } Replica_1 : A_1 = \{a\},\ R_1 = \{a\},$$

$$\text{State of } Replica_2 : A_2 = \{a, b\},\ R_2 = \emptyset.$$

When merged:

$$A_{merged} = \{a, b\},\ R_{merged} = \{a\}.$$

The resulting set is:

$$A_{merged} \setminus R_{merged} = \{b\}.$$

3. LWW-Set (Last Writer Wins Set)

In the Last Writer Wins Set, each element has a `timestamp` and a `visibility flag`. The most recent update (based on timestamp t) determines the element's state [56, 47].

Example:

- At $Replica_1$: $add(a, t_1)$, where $t_1 = 1$

- At $Replica_2$: $remove(a, t_2)$, where $t_2 = 2$

The states at the replicas are:

$$\text{State of } Replica_1 : \{(a, t_1, \text{true})\},$$
$$\text{State of } Replica_2 : \{(a, t_2, \text{false})\}.$$

When merged:

$$\{(a, t_2, \text{false})\}.$$

The final state removes a, since $t_2 > t_1$.

4. PN-Set (Positive-Negative Set)

A PN-Set uses counters to track `adding` (P) and `removing` (N). An element is in the set if $P > N$ [56, 47].

Example:

- At R_1: $add(a)$, $remove(a)$
- At R_2: $add(a)$, $add(a)$, $add(b)$

The counters at the replicas are:

$$\text{State of } Replica_1 : P_1 = \{a : 1\},\ N_1 = \{a : 1\},$$
$$\text{State of } Replica_2 : P_2 = \{a : 2, b : 1\},\ N_2 = \emptyset.$$

When merged:

$$P_{merged} = \{a : 2, b : 1\},\ N_{merged} = \{a : 1\}.$$

The resulting set is:

$$\{a, b\},\ \text{where } P > N.$$

5. OR-Set (Observed-Remove Set)

An OR-Set uses unique tags to track `adding` and `removing` for each element. An element is in the set if it has tags that have not been removed [56, 47].

- T: Set of tags added.
- R: Set of tags removed.

Example:

- At $Replica_1$: $add(a, t_1)$, $add(b, t_2)$, $remove(a, t_1)$
- At $Replica_2$: $add(a, t_3)$, $add(c, t_4)$

The states at the replicas are:

$$\text{State of } Replica_1 : T_1 = \{a : \{t_1\}, b : \{t_2\}\},\ R_1 = \{a : \{t_1\}\},$$
$$\text{State of } Replica_2 : T_2 = \{a : \{t_3\}, c : \{t_4\}\},\ R_2 = \emptyset.$$

When merged:

$$T_{merged} = \{a : \{t_1, t_3\}, b : \{t_2\}, c : \{t_4\}\},\ R_{merged} = \{a : \{t_1\}\}.$$

The resulting set is:

$$\{a, b, c\},\ \text{where } T \cap R = \emptyset.$$

2.3.4 Simple CRDT implementation above SQL DB

The `G-Set` and `LWW-Map` structures, which are minor modifications of the LWW-Set 2.3.3, can be utilized to synchronize data across different devices. Consequently, the table will function as a `G-Set` of `LWW-Maps`. To manage deletions, it is essential to introduce a specific attribute, such as `tombstone`, to indicate that an element has been deleted. Each update will result in multiple entries in the table (Table 2.1); for instance, changing a name and type will generate two separate records in the table [44]. Based on these concepts, several libraries have been developed, including Evolu[3] and crdt[4]. Since this was considered an elegant idea, it was included in the thesis. However, the next part will focus on Automerge[5], which is unrelated to this concept.

id	timestamp	dataset	row_id	column	value
1	...	accounts	e29d69a6-148e-...	name	S:Checking
2	...	accounts	e29d69a6-148e-...	type	S:checking
3	...	accounts	e29d69a6-148e-...	offbudget	N:0
4	...	accounts	e29d69a6-148e-...	closed	N:0
5	...	payees	503189fc-efc1-...	name	S:
6	...	payees	503189fc-efc1-...	transfer_a	S:e29d69a6

Table 2.1: Illustration of a simple CRDT table used in a SQL database for a money tracking application. The `dataset` indicates which table the update refers to. The combination of `row_id`, `column`, and `value` specifies the exact cell that was modified and its new value. The `timestamp` shows when the update occurred, which is crucial for accurately merging different updates.

2.3.5 Automerge

Automerge[6] is a library that implements complex CRDT types. Some of its authors proved eventual consistency for some CRDTs [28]. This library shows basic principles of how complex data can be shared. For example, for the action in Listing 2.1, Automerge will generate these logs in Listing 2.2 that will be synchronized to achieve the CRDT attributes [41].

```
state = Automerge.change(state, "Add todo item",
    (doc) => {
        doc.todos.push({
            title: "Buy milk",
            done: false
        })
    })
```

Listing 2.1: Example of adding a todo item in Automerge.

[3] https://github.com/evoluhq/evolu
[4] https://pub.dev/packages/crdt
[5] https://github.com/automerge/automerge
[6] https://github.com/automerge/automerge

```
{action: "makeMap", obj: id1}
{action: "set", obj: id1, key: "title", value: "Buy milk"}
{action: "set", obj: id1, key: "done", value: false}
{action: "ins", obj: todosID, key: prevID, elem: 15}
{action: "link", obj: todosID, key: elem15, value: id1}
```

Listing 2.2: Change log for adding a todo item that is then synced across clients to achieve eventual consistency.

2.3.6 CRDT's In Text Editing

A good example of text editing is Google Docs. In Google Docs, users edit the same document in real time and even offline. When user changes are merged, we want to keep both user changes (Figure 2.1). The important thing is that this insertion does not depend on the actual position index but rather on the relative position to other elements. It can also handle inserting at the same position from multiple clients. It then uses its indexes to merge it automatically to be the same on all clients [41]. The correctness of this algorithm was formally proved for Automerge [28].

Another popular library is Yjs[7] which is used for apps such as: Evernote[8], AppFlowy[9], GitBook[10], Amazon SageMaker[11]. Yjs is also used as one of the storage formats for the very popular text editor library TipTap[12], which can be combined with PowerSync[13] to have a collaborative editor on the web [53].

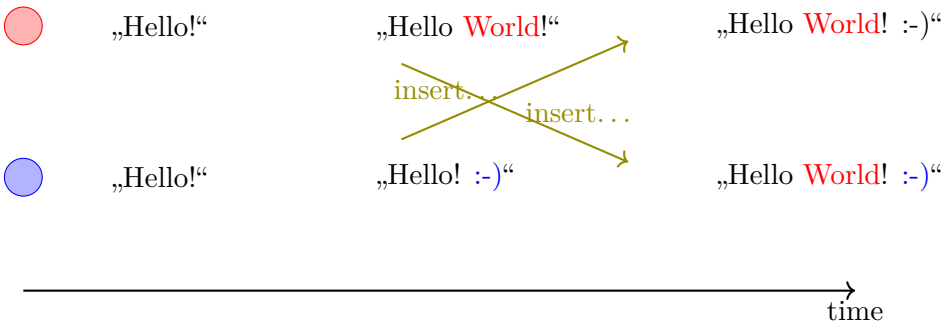


Figure 2.1: Diagram showing that after clients exchange text-editing changes, it should end up in the same state.

2.4 Existing Time-Management Applications

The production apps keep their synchronization implementation details private. So the best real-world inspirations are open-source time-management apps such as AnyType[14] or

[7] https://github.com/yjs/yjs
[8] https://evernote.com/
[9] https://github.com/AppFlowy-IO/AppFlowy
[10] https://www.gitbook.com/
[11] https://aws.amazon.com/sagemaker/
[12] https://github.com/ueberdosis/tiptap
[13] https://www.powersync.com/
[14] https://anytype.io/

AppFlowy[15]. They allow you to share complex data across multiple users. Complex data is meant by the content of text editors, where it is crucial to combine inputs from multiple users in a predictive way. Good examples are conflict-free replicated data types (CRDTs) mentioned in Section 2.2, designed to help solve these problems.

2.4.1 AnyType – Existing Time-Management Application

AnyType uses its own synchronization mechanism called `any-sync` [5]. This system is designed to work by default without needing an external server, allowing for peer-to-peer communication over a local network. It also employs Conflict-free Replicated Data Types (CRDTs) to ensure that data is synchronized consistently. However, `any-sync` is only utilized in their open-source project[16]. AnyType's primary goal is to offer software that operates with "no one in between." In practice, though, for data to be synchronized across multiple devices that are not open at the same time, a synchronization server is still required. In AnyType's case, this can be a server provided by AnyType or a self-hosted server[17].

Media files are not automatically downloaded during synchronization to conserve bandwidth. Instead, they are streamed from a backup node or other devices on the network when requested. For example, images download immediately upon being opened, while videos or audio files start downloading as they are played. Additionally, AnyType features a deduplication process that helps minimize storage usage [6]. For instance, if the same image is uploaded multiple times, only one copy is stored, which saves space.

2.4.2 AppFlowy – Existing Time-Management Application

AppFlowy is an excellent case study for exploration because it is developed using Flutter and Rust for the front end, which aligns with the TimeNoder2 implementation. Its development began in 2021, a time when Flutter was relatively new, making the combination of Flutter and Rust a safer and more modular approach to building the app. This way, even if Flutter were to become outdated, a significant amount of the code could still be reused [10].

AppFlowy follows domain-driven design (DDD) architecture for app development, which includes four layers: presentation, application, domain, and infrastructure, as shown in Figure 2.2 [37]. In AppFlowy, the infrastructure layer is built with Rust, while the other layers use Flutter [10]. The use of Rust for the infrastructure layer allowed AppFlowy to later incorporate a front-end Tauri[18] implementation to enhance the desktop user experience[19]. For back-end synchronization, users can utilize AppFlowy Cloud[20], or they have the option to self-host it using Supabase[21].

In the front-end Rust component, AppFlowy utilizes Diesel[22] Rust DB to manage user data [9]. The Rust side of AppFlowy communicates with Flutter through the `FFI`[23] package designed for Flutter. This communication is achieved using interfaces defined in Flutter and implemented in Rust, as illustrated in Figure 2.3.

[15] https://github.com/AppFlowy-IO/AppFlowy
[16] https://github.com/anyproto/any-sync/network/dependents
[17] https://tech.anytype.io/how-to/self-hosting
[18] https://github.com/tauri-apps/tauri
[19] https://github.com/AppFlowy-IO/AppFlowy/discussions/1746
[20] https://github.com/AppFlowy-IO/AppFlowy-Cloud
[21] https://supabase.com/
[22] https://github.com/diesel-rs/diesel
[23] https://dart.dev/interop/c-interop

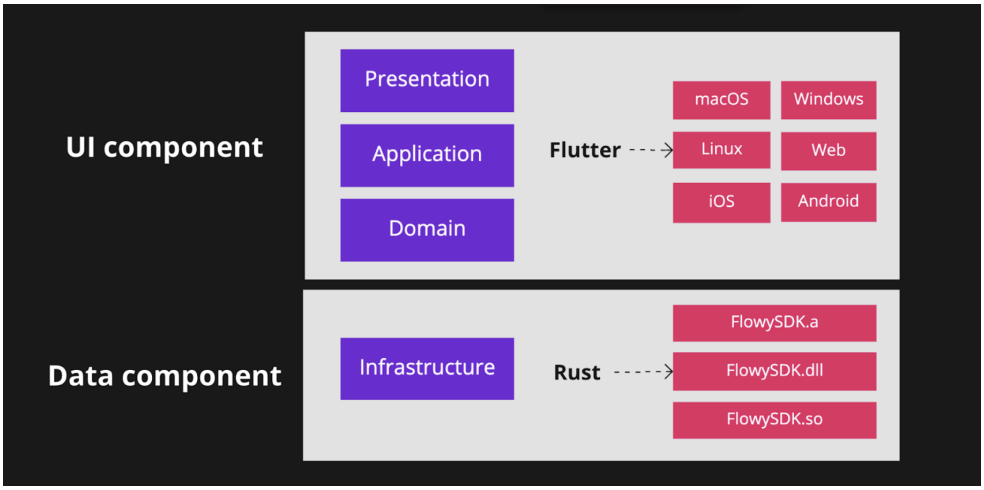


Figure 2.2: AppFlowy Domain-driven design (DDD) architecture diagram.

To clarify this concept, consider the following example. When a user makes changes to a document, such as adding new text, the text editor creates a `transaction` in Flutter. This `transaction` captures the details of the changes made to the document. The application then invokes a function in Flutter's interface, which may be termed as *applyChangesOnDocument(documentId, transaction)*. This function subsequently calls a corresponding function on the Rust side, which applies the changes to the local Diesel[24] database. The data is encoded using yrs[25], a library for Conflict-free Replicated Data Types (CRDTs), enabling it to be shared among multiple devices [7].

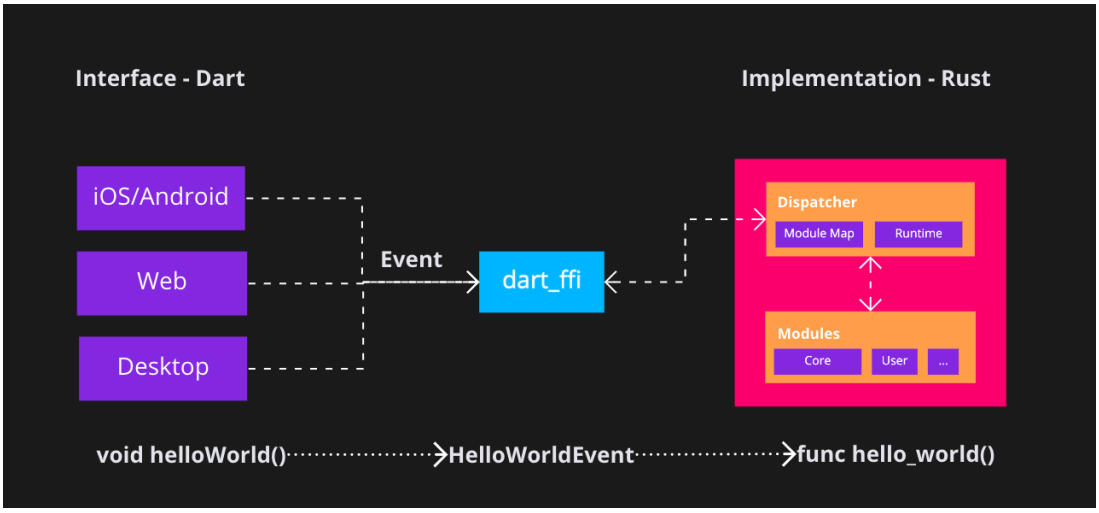


Figure 2.3: Diagram showing how AppFlowy handles communication between Rust and Flutter.

When a user chooses to sync AppFlowy with a self-hosted Supabase, the application utilizes Supabase's real-time functionality through Flutter to listen for changes in the backend [8]. When updates occur in the database, the system notifies the Rust front-end to

[24] https://diesel.rs/
[25] https://github.com/y-crdt/y-crdt

prompt an update. The Rust component then uses the `supabase-rs`[26] library to communicate with Supabase and retrieve the latest changes[27].

2.5 Existing Synchronization Libraries

Synchronization has been a long-standing issue, leading to the development of various solutions. Some solutions are tailored specifically for Flutter and can be integrated directly, while others are geared towards web applications and can serve as creative inspirations. The solutions discussed here do not utilize CRDT (Conflict-free Replicated Data Types) principles; instead, they focus on tracking changes. Most of these solutions rely on SQL databases for storage and address conflicts at the level of table rows [64, 54]. This method differs from CRDTs, which enable conflict resolution at the individual value level [56].

Nevertheless, it is feasible to integrate CRDTs into these systems. For example, CRDT types can be converted into a binary format and stored as an attribute in a table. This binary data can be transformed back into CRDT format on the client side. By merging these methods, it becomes possible to develop a collaborative text editor [53]. This same approach is also utilized in AppFlowy, as previously mentioned [7].

2.5.1 PowerSync

PowerSync[28] has recently become a popular choice for synchronization solutions. Although it is a paid service, a self-hosted version is also available with some limitations. PowerSync provides software development kits (SDKs) for several platforms, including Flutter, React Native, Web, Kotlin, and Swift[29]. To set up PowerSync, a PostgreSQL database, a PowerSync server, and a local device that connects to the server are required. Data from PostgreSQL is synced to the user's device based on defined sync rules[30]. PowerSync is capable of synchronizing data for individual users (Listing 1) as well as for teams (Listing 2). Additionally, it supports the creation of a client-side schema that is customized for a specific SDK according to these sync rules (Listing 3).

```
bucket:
    user_data:
        # (request.user_id() comes from the JWT token)
        parameters: SELECT request.user_id() AS user_id
        data:
            - SELECT * FROM lists WHERE owner_id = bucket.user_id
```

Listing 1: By adding a `user_id` attribute to each record that uniquely identifies a user, similar synchronization rules can be applied, as shown in the example, to ensure that user data is consistent across devices.

[26] https://github.com/supabase-community/postgrest-rs
[27] https://github.com/supabase-community/postgrest-rs/network/dependents?dependents_after=MzE4NTE2Mjk3OTI
[28] https://www.powersync.com/
[29] https://docs.powersync.com/intro/powersync-overview
[30] https://docs.powersync.com/usage/sync-rules

```
bucket:
    team_data:
        # request.user_id() comes from the JWT token
        parameters:
            SELECT account_id FROM basejump.account_user
            WHERE user_id = request.user_id()
        data:
            - SELECT * FROM project
              WHERE account_id = bucket.account_id
```

Listing 2: Rules for synchronizing user and team interactions with the Supabase backend, based on the Basejump Supabase template[31].

```
const schema = Schema([
    Table('todos', [
        Column.text('list_id'),
        Column.text('created_at'),
        Column.text('description'),
        Column.integer('completed')
    ])
]);
```

Listing 3: Example of a Flutter client-side schema generated by PowerSync for a todo application.

2.5.2 WatermelonDB

WatermelonDB[32] is a local database designed for React Native, which means it cannot be used directly with Flutter. However, it serves as a useful example for implementing synchronization between the backend and frontend. WatermelonDB has established a protocol for synchronizing the local database with a server that follows this protocol [62]. This synchronization process requires the server to implement two functions: *push* and *pull*. Each synchronization step involves the client first calling *pull* to retrieve data from the server, followed by calling *push* to send updated data back to the server as seen in the Figure 2.4.

- The *pull* function sends a `last_pulled_at` timestamp to the server, which then responds with the `changes` that have occurred since that `last_pulled_at` time.
- The *push* function sends also a `last_pulled_at` time, but this time referring to the start of the synchronization. It is sent along with changes from the local database that are newer than the server's. The server checks for any `changes` that happened since that `last_pulled_at` time. If there are any `changes` detected between the *push* and *pull*, the *push* will be stopped, and an error will be sent back to the client. This error notifies the synchronization application to attempt the synchronization process again. If no differences are found between the *push* and *pull* actions, the `changes` will be successfully added to the database.

[32] https://github.com/Nozbe/WatermelonDB

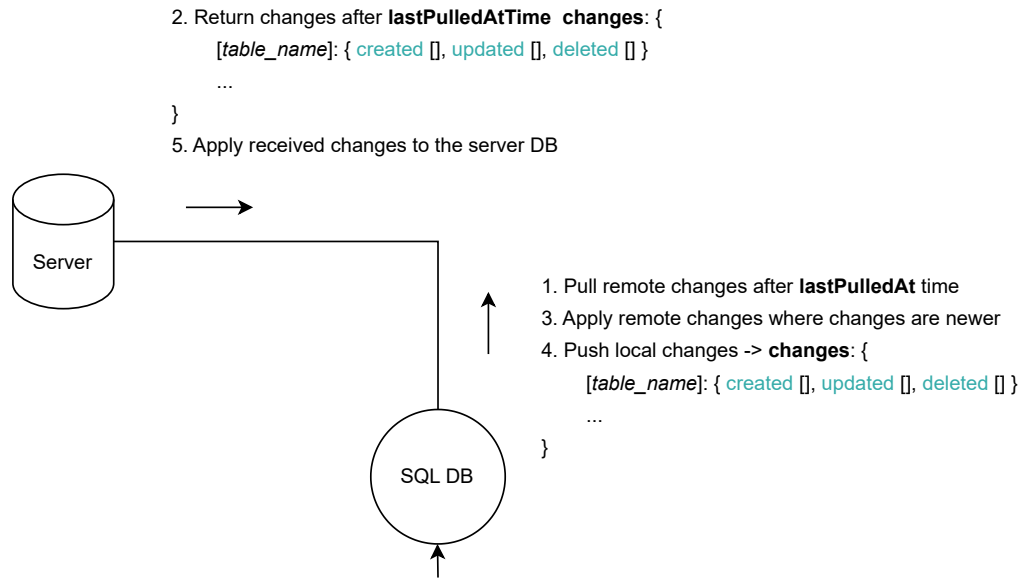


Figure 2.4: Watermelon DB synchronization diagram showing the protocol timeline.

There is a good example of how to implement this using Supabase, a popular Backend-as-a-Service (BaaS) platform. BaaS provides essential features such as authentication, database management, and file storage, allowing developers to use it in place of traditional backend servers [23]. This example includes two Supabase database functions, *push* and *pull*, which operate within a transaction. These functions can interact directly with specified tables and their attributes [49], or they can be designed to be generic [20]. The benefit of the generic approach is that adding a new table to the database does not require changes to the *push* and *pull* code on the backend.

The implementation necessitates the storage of additional attributes [49]. The attributes `updated_at` and `created_at` are utilized by WatermelonDB to identify what needs to be sent to the backend for synchronization. In addition to these attributes, WatermelonDB tracks deleted items using the *somePost.markAsDeleted()* function [63], which determines which `ids` of deleted items will be included in `changes`. On the server side, attributes such as `deleted_at`, `server_created_at`, and `last_modified_at` are employed to identify what should be sent back to the client via the *pull* function and what updates should be made to the backend database within the *push* function. The specific attributes used for synchronization on the server may vary according to the chosen implementation of the database functions, although they will generally resemble those previously mentioned.

Chapter 3

Smartwatch Operating Systems

While the Apple Watch maintains its lead, HarmonyOS and WearOS are witnessing growth. In China, HarmonyOS's share in the smartwatch market (Figure 3.1) is projected to hit 61%, driven by the widespread use of Huawei's 5G smartphones. The global smartwatch market is also on the rise, with a surge in basic models that run proprietary operating systems, featuring fundamental functions and applications. It is important to differentiate smartwatches with high-level operating systems (HLOS), primarily produced by Apple and Samsung. The market is anticipated to grow by 14% in 2024, largely due to the expansion of WearOS and HarmonyOS devices. Furthermore, companies like OnePlus, OPPO, and Xiaomi are increasingly focusing on launching WearOS watches outside China at appealing premium price points (Figure 3.2) [21].

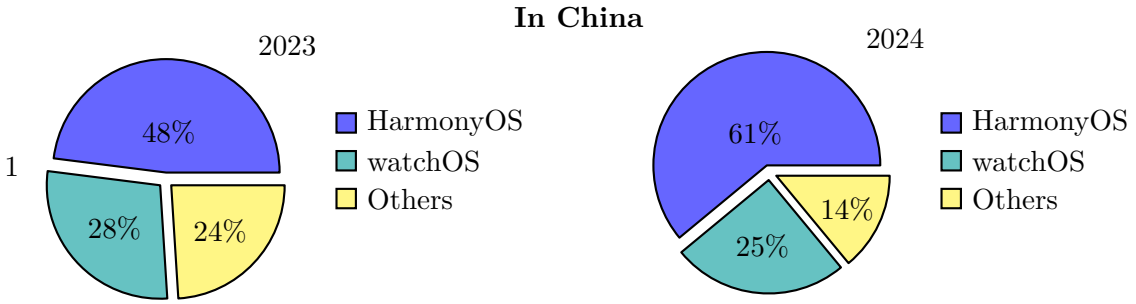


Figure 3.1: Smartwatch shipment share in China for 2023 and 2024 [21].

Figure 3.2: Smartwatch shipment share outside China for 2023 and 2024 [21].

Smartwatches are used mainly in fitness and health tracking. The Apple Watch is a promising wearable device for health monitoring, particularly in mental health. It effec-

tively tracks physiological parameters and wellness. Research indicates that metrics such as heart rate variability (HRV) correlate with changes in emotional and physical states. By integrating data from various sensors on activity and sleep, along with user inputs, the device helps to monitor and potentially diagnose mental health disorders [45].

This is supported by another study that created its own prediction models based on Apple Watch ECG sensor data [58]. But is important to say that the studies in this area are also done on other devices (Android based, custom build, etc.) [39]. However, for more complex AI models, selecting the appropriate architecture is crucial, as executing these predictions on smartwatches may result in high memory usage, while cloud-based computations could be hindered by internet connectivity issues [65].

A significant case [31] highlighting the role of Apple Watches involved a 56-year-old individual. Although asymptomatic at first, he detected an unusually elevated heart rate via his Apple Watch, corresponding to palpitations. Four days afterward, an electrocardiogram (ECG) validated a diagnosis of Atrial Flutter. He subsequently received anticoagulation therapy and underwent electrical cardioversion.

A substantial study with 419,297 participants evaluated the Apple Watch's effectiveness in detecting atrial fibrillation (AF) over eight months. Results showed that 0.5% of participants had an AF episode over 30 seconds, with the incidence rising to 3.2% in those 65 years and older. Additionally, compared to ECG patch assessments, the watch exhibited a positive predictive value of 0.84, highlighting its potential benefits in medical practice [51]. Certain limitations must nonetheless be acknowledged. Firstly, the potential overdiagnosis of AF poses a concern, as it may lead to undue anxiety [31].

For middle-aged and older patients, the use of activity monitors increased the effectiveness of weight loss programs [18]. For older adults, falls are a leading cause of injuries, so recognizing these events is vital. Smartwatches have the capability to aid in identifying such occurrences [59]. The Apple Watch 6 struggles with accurate oxygen saturation readings compared to medical pulse oximeters, with many failures, especially in children due to movement [50]. The accuracy of step count and heart rate measurements was evaluated, and the results were found to be approximately correct, even for watches from around 2017 [12]. Another study supports the good accuracy of the step count measurement [17].

Smartwatches can also be used in education. In a study that combined English lessons with exercise, students performed better on tasks and developed positive feelings toward both the class and the devices [55]. Another study explores using smartwatches to help primary school students reflect on their science learning. Using smartwatches, students could think about what they learned in real-world situations. This method helped improve their confidence in performing science tasks, making them feel more capable and motivated [27].

Construction projects expose workers to hazardous environments, making the construction sector one of the highest-risk industries in the USA. It accounts for 21.4% of fatalities across all industries. Additionally, the construction sector has the second-highest number of work-related severe injuries, accounting for 19% of hospitalizations and 10% of amputations. Smartwatches can assist in improving safety by monitoring physiological data for emotional and stress assessment [2]. This is supported by another study [1].

Wearable technology for health monitoring raises ethical issues, particularly regarding privacy. The data, useful for medical purposes, can also be exploited for marketing. Data brokers make money by selling personal health information collected from apps and wearables, which track real-time locations, activities, and behaviors. Many health apps or devices, especially free or low-cost ones, use a business model that relies on selling user data [26].

3.1 Smartwatch UI and Usability

Understanding the user's needs is crucial to ensure that an app works effectively. The strategy may vary significantly depending on the platform, making it essential to become familiar with the best UI/UX practices for each platform [42]. This section will explore expert advice on UI/UX when designing smartwatch applications.

Smartwatches are often used alongside other devices, so they need to function well together. This raises a significant question: Which device is best for specific tasks, and how can they enhance each other's performance? Using smartwatches differs from using other devices. For example, scheduling applications such as Google Calendar on desktops provide extensive control and information, making them suitable for complex tasks. This is generally true for desktop applications [34]. Research [32] indicates that while people spend less time on desktops, they engage in longer sessions compared to mobile apps, which tend to be used more frequently but for shorter tasks. A similar relationship can be observed between mobile apps and smartwatches, as illustrated in Figure 3.3 [19, 15, 15].

Figure 3.3: Illustration of utilizing Google Maps on both a smartphone and a smartwatch [16].

Smartwatch interactions usually take about 5 seconds [14] compared to 3 minutes and 52 seconds on the phone [60]. But people check their watches over 150 times a day [4]. Because of this, smartwatch apps should focus on being quick and easy to use, such as showing what is coming up next on a calendar. Google recommends a three-layer pyramid design for smartwatch use: wear as the base, glance in the middle, and interact on top [15].

Since watches are mostly worn, it is important to add value through features that don't need interaction, like tracking your heart rate. The glance layer is great for quickly helping

users. People look at their watches briefly to track fitness or see notifications, so this layer should show important information at the right time, allowing users to quickly decide whether to act or wait, while keeping distractions to a minimum. The interaction layer is for less frequent and longer use. When people use this layer, it should be easy to reach their goals with a few taps, swipes, or button presses [15].

3.1.1 WearOS

WearOS has different ways to improve how users interact, such as using apps, tiles, notifications, and complications. The watch face is central to the experience and can be personalized to match the user's style. It can include complications for quick data access, such as steps walked or heart rate, as shown in Figure 3.4. Tiles are interactive options users can reach by swiping to the sides from the watch face. They provide easy access to important information and actions as illustrated in Figure 3.5. Users can choose and arrange the tiles they want in a carousel. Notifications give quick alerts, helping users get important information quickly. They should be short and well organized for easy reading, as shown in Figure 3.6 [15].

Figure 3.4: Examples of complications show in various watch faces, such as walked a number of steps, heart rate, actual date, or current temperature [4].

Figure 3.5: Tile examples concentrate on the primary, straightforward tasks that users wish to accomplish [4].

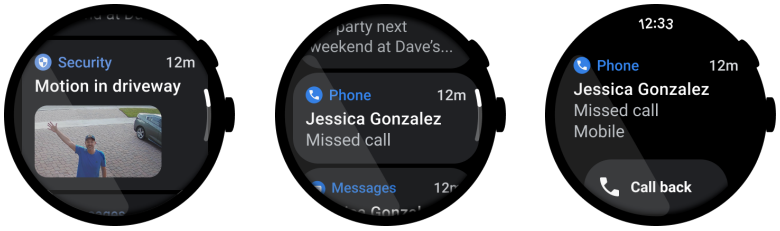


Figure 3.6: Tile examples helping users get important information quickly [4].

Apps allow users to do more complicated tasks that require additional actions, such as tapping and scrolling as depicted in Figure 3.7. Users can access these apps directly from the watch face, tiles, notifications, or the app launcher. Some apps, like those used for tracking workouts, need to run all the time, which is called ongoing activities. WearOS makes these activities easy to reach by offering entry points on the main interfaces such as the watch face, tiles, and launcher. In addition, the content, such as weather information in WearOS is flexible and can appear in different places, such as complications, tiles, or apps, as shown in Figure 3.8 [15].

Figure 3.7: Downloading a song in a music app [4].

(a) Complication (b) Tile (c) App

Figure 3.8: Example of how an application can scale across surfaces [15].

When creating apps for WearOS, it is important to focus on just a couple of important things users need, not the whole app. Design for the wrist so tasks are fast and easy and do not tire the user's arm because a user is expected to be 'somewhat strongly' tired after only 3 minutes of use of the smartwatch in a standing pose, and 4 minutes of use when sitting [38]. Ensure the interface is simple and works well with different watch faces and complications. Let the smartwatch be easily connected to phones and other devices, deciding which tasks are best for each. Also, add offline features so users can still do things without always needing a phone connection [15].

3.1.2 Apple Watch

When developing for the Apple Watch, the process is somewhat similar to that of WearOS, but there are some key differences. Apple emphasizes a more cohesive design system that is focused on consistency in its applications. To help developers create great apps, Apple provides ready-made templates that they can easily use, as illustrated in Figure 3.9 [11].

Developing for the Apple Watch involves thoroughly understanding these design principles and the tools available. However, a detailed exploration of these principles and tools will not be included in this thesis, as the goal of this thesis is to target WearOS.

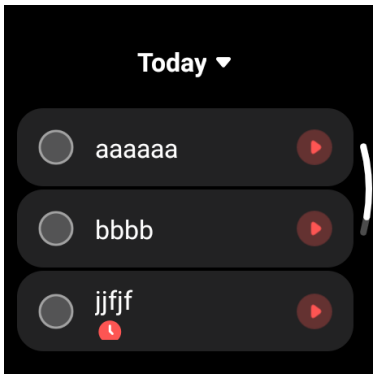
Figure 3.9: Apple Watch foundational layouts [11].

3.1.3 Examples of Applications in the Context of TimeNoder2

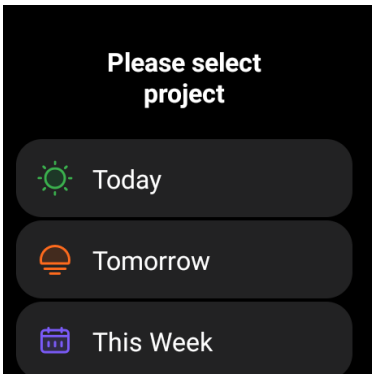
Based on research and earlier studies, this work examines the smartwatch applications of Todoist and Focus Todo [69].

Focus Todo

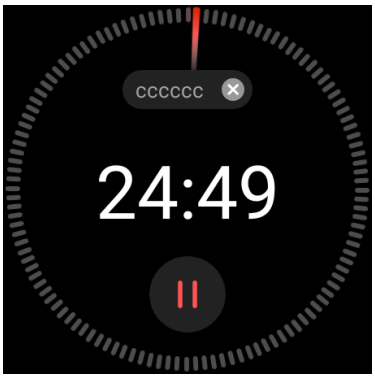
Focus Todo is an app that combines a to-do list with the Pomodoro technique [69]. They can track or complete tasks and check the time they have tracked for the day. The smartwatch must stay connected to the phone for the app to work properly. For example, if the smartwatch disconnects from the phone, all changes made on the smartwatch will not be saved. A similar problem can be found in the TickTick smartwatch app. The Focus Todo smartwatch app also does not allow users to add tasks.



(a) Today's tasks: the user can track or complete them.

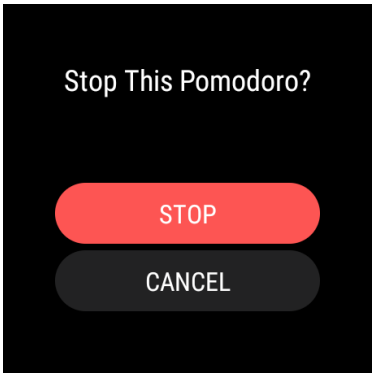


(b) View for selecting a time-based or classic project.



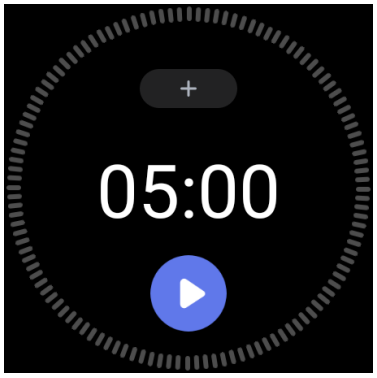
(c) Pomodoro tracking of work-time.

(a) Pomodoro tracking of work-time in ambient mode.

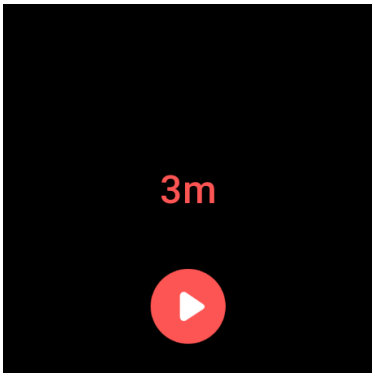


(b) Dialog to confirm cancellation of tracking.

(c) Pomodoro tracking statistics for the current day.



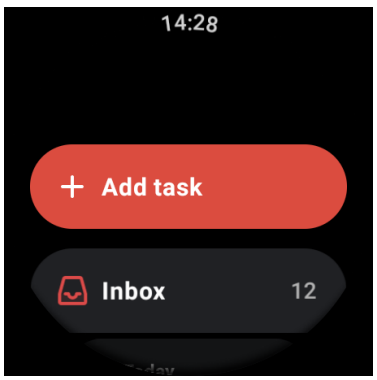
(a) Pomodoro tracking of break-time.



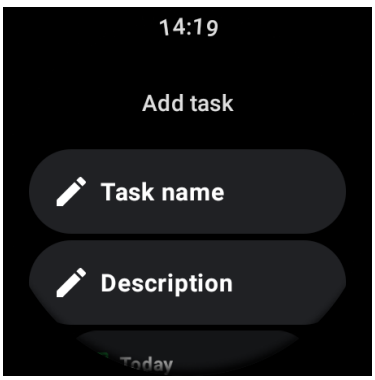
(b) Tile for entering pomodoro tracking.

Todoist

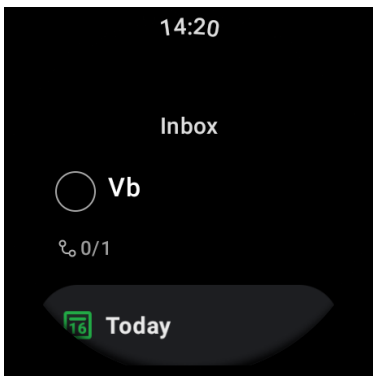
Todoist aims to provide an effective to-do list solution. The smartwatch app allows users to view their tasks, mark them as complete, adjust task deadlines, and add sub-tasks. A significant advantage of this application is that users do not need to be connected to their phones for it to function correctly.



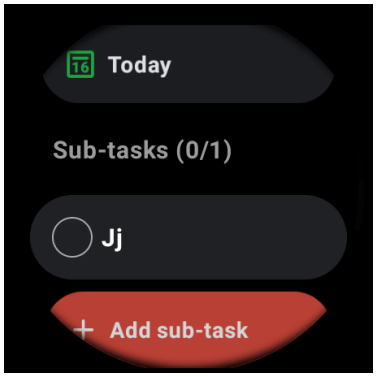
(a) The home screen shows time-based and classic projects.



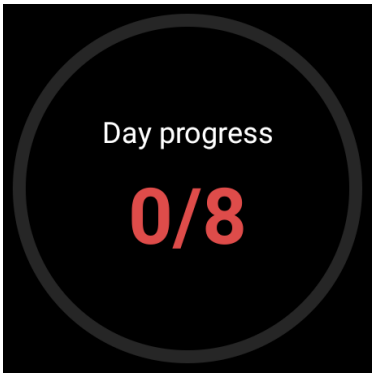
(b) Creating a task dialog with options to select name, etc.



(c) Task details part 1: complete task or change date.



(a) Task details part 2: add subtask, complete subtask.



(b) Tile showing the number of completed tasks today.

3.2 Communication Between a Phone and a Smartwatch

Smartwatches primarily use Bluetooth to communicate with smartphones. This connection enables them to transfer data efficiently, relying on the phone for internet access. When a smartwatch is not connected to a phone, it can use direct Wi-Fi access to connect to the internet. However, switching between Bluetooth and direct WiFi is key for smooth operation. There are challenges with Bluetooth, such as connection issues and delays that arise from relying on a paired phone. This can negatively impact the user experience. Therefore, choosing the right connection, Bluetooth or WiFi, is really important. A poor choice can lead to performance problems, and switching between the two can disrupt activities [68].

Wearable devices, such as smartwatches and fitness trackers, also share sensitive health information with smartphones via Bluetooth. Research indicates that even when these data are encrypted, they can still be at risk from eavesdroppers. These individuals may be able to identify which devices are being used and monitor user activities, such as noting an insulin injection. This presents serious privacy issues. Common security measures, like introducing delays or using fake data, have not been very effective and can slow down performance. Thus, there is a clear need for better methods to protect sensitive information and limit unnecessary data sharing in wearables, ensuring both user privacy and a smooth experience [13].

Networking and sending information on WearOS smartwatches can quickly drain their smaller batteries [36]. To help, avoid using the internet for non-essential tasks unless the watch is charging. Instead, let your mobile device handle more intensive data tasks, such as synchronization, when possible. In this way, the smartwatch can be updated when connected to Wi-Fi and charging, helping to conserve battery life while still maintaining functionality [3].

Sharing of Data Between WearOS Smartwatches and Phones

Operating systems provide native synchronization mechanisms to facilitate communication between smartwatches and smartphones. For Android devices, the Data Layer API represents the primary and recommended method for data exchange between a WearOS application and its companion Android application [30]. This API, integrated within Google Play services, enables various communication patterns, such as message transmission and asset transfer (including images and other files).

It is important to note that this communication paradigm has specific requirements. First, communication is limited to Android phones; this mechanism does not support iOS devices. Second, both the WearOS application and its companion phone application must share identical bundle identifiers and signature keys to establish a secure connection [25].

After fulfilling these prerequisites and establishing a Bluetooth connection between the smartphone and the WearOS device, the Data Layer API enables bi-directional communication. For Flutter developers, the `watch_connectivity` plugin offers a convenient wrapper around these native APIs, as demonstrated in Listing 4. This abstraction simplifies the implementation of cross-device communications. It is also important to account for the underlying constraint that individual messages cannot exceed 100KB in size [29].

```
// Send data from phone to watch or vice versa
watch.sendMessage({'data': 'Hello'});

// Listen for incoming messages
watch.messageStream.listen((message) {...});
```

Listing 4: The `watch_connectivity` plugin provides methods for bidirectional communication between WearOS smartwatch applications and their companion Android phone applications built using Flutter.

An alternative synchronization approach leverages the smartwatch's direct internet connectivity capabilities, either through Wi-Fi or cellular connections, or via Bluetooth-tethered connectivity to the paired phone. This strategy allows WearOS applications developed in Flutter to implement identical synchronization mechanisms as those used in other platforms of cross-platform applications. This approach offers significant advantages, particularly the ability for smartwatches to synchronize with cloud services independently, without requiring a constant connection to their companion phones. For instance, in applications using synchronization frameworks like PowerSync (discussed in section 2.5), developers can reuse the same synchronization code across all platforms, including WearOS smartwatches, resulting in more consistent behavior and simplified maintenance.

Chapter 4

Analysis of TimeNoder2 Architecture and Synchronization Support

4.1 Overview of TimeNoder2

TimeNoder2 builds on the groundwork laid in the bachelor's thesis, during which the preliminary version, TimeNoder1, was created [69]. This chapter delves into the app's progression, existing functionalities, and avenues for improvement. The bachelor's thesis introduced TimeNoder1 as a holistic time management solution, drawing on thorough theoretical insights from time management theory. It combined multiple time management techniques within one app. Feedback from both the thesis opponent and initial users revealed that, although feature-rich, TimeNoder1 compromised usability.

New users struggled with the interface due to layered nesting, complex hierarchy, and multi-step processes for straightforward tasks, which led to a higher cognitive load. As a result, the application was completely restructured. This revised application underwent validation by the thesis opponent. Due to the limited user base at that stage, cross-platform compatibility was not prioritized. The following highlights the key differences between TimeNoder1 and TimeNoder2.

Features Added in TimeNoder2:

- Enhanced UI/UX.
- Option to share app data copies across devices.
- New features including the Eisenhower Matrix, rewards, and habits.
- Android task widget.
- Broad customization options (sounds, themes).
- Ability to link multiple tasks and projects to events.

Features Removed in TimeNoder2:

- Synchronization across devices for individual users.
- Complex planning features, which previously complicated and slowed the app.

These modifications have driven TimeNoder2's growing popularity, evidenced by the following metrics:

- **Downloads:** 9,590,
- **User Ratings:** 4.728 average, based on 261 reviews, 148 with comments,
- **Community Articles:** Widespread discussions on platforms like Reddit reflect strong engagement:
  - *Apps of 2023: What are your best and worst discoveries of 2023?*[1],
  - *TN2: TimeNoder2: Mächtige, kostenlose Produktivitäts-App für Android und iOS*[2],
  - *TimeNoder2: une puissante application de productivité gratuite pour iOS et Android*[3].
- **Stores:** Android[4], iOS[5],
- **TimeNoder2 Discord Server:** 284 members[6],
- **YouTube Tutorial**[7].

The redesign tackled TimeNoder1's deep nesting by simplifying the data model and flattening navigation, while retaining Material Design principles to ensure a consistent user experience.

4.2 User Feedback and Requirements

The TimeNoder2 app has been reviewed 261 times on Google Play, and 148 reviews include comments. Although brief remarks like *"Great app!"* or *"Top!"* provide limited insight, more elaborate responses have been collected for analysis. Users consistently praise the app's extensive features and intuitive design (UI/UX), which is mentioned in 32 reviews. The application's customizability is valued in 11 instances, while calendar integration and developer responsiveness are noted 4 times each. Users also commend specific techniques – such as timeboxing (10 mentions), Pomodoro (4 mentions), as well as the Eisenhower Matrix (3 mentions) and reminders (3 mentions). Suggested improvements include localization (13 mentions), cross-platform support or syncing (10 mentions), and enhanced widgets

[1] https://www.reddit.com/r/androidapps/comments/18ptm4b/apps_of_2023_what_are_your_best_and_worst/
[2] https://stadt-bremerhaven.de/tn2-timenoder2-maechtige-kostenlose-produktivitaets-app-fuer-android-und-ios/
[3] https://infoidevice.fr/timenoder2-application-productivite-ios-android/
[4] https://play.google.com/store/apps/details?id=com.janzimola.goal_venture2&hl=en
[5] https://apps.apple.com/us/app/timenoder2-timeboxing-master/id6468406842
[6] https://discord.gg/e8b47EvCeq
[7] https://youtu.be/psB7zraKl44?si=KJbDBc0r72euRHji

(3 mentions). Problems reported involve notifications (5 mentions), crashes (4 mentions), timer tracking and widget functionality. The complexity arising from an abundance of features is highlighted in 5 reviews. Comparisons with competitors like TickTick (4 mentions), Amazing Marvin, Todoist, Focus Todo, and Engrow (1 mention each) are common.

4.2.1 Need for Cross-Platform Support

While many users appreciate that TimeNoder2 is both free and devoid of ads, there is a notable interest in paying for enhanced features. On the Discord server, the request for cross-platform compatibility supported by a subscription option is the second-highest demand, receiving 15 endorsements of *"definitely for it."* However, this feedback reflects only part of the interest, as not all users participate on Discord. A Reddit post celebrating TimeNoder2 as a standout in 2023 earned 403 upvotes, with the app receiving 119. Additionally, a comment highlighting the absence of cross-platform support as a barrier for those considering a switch from TickTick earned eight upvotes, indicating potential for user conversion. The enthusiasm for this feature and willingness to pay suggest that offering it at a competitive price could set TimeNoder2 apart from established competitors like TickTick.

4.2.2 Other User Needs

Localization tops feature requests, with volunteers offering translation help. Bug fixes for widgets, notifications, and timers are urgent. Although there are plenty of feature suggestions, managing complexity is vital to retain users, as some reviews call the app powerful yet complex. Material Design's success in UI/UX feedback suggests no need for a design overhaul.

4.2.3 Analyzing Discord Voting

Members of the app's Discord server vote for features in a special channel. They can choose ratings from 1 (not wanted) to 5 (very important). Prioritizing 5 and 4 ratings avoids skew from older, less visible options. The item notation consists of a **feature name** – a short description, and feature ratings. Feature ratings are a list of pairs, where the first item of the pair is the number of times the rating occurred, and the second is the rating on the scale of 1 to 5. The notation of the rating is defined by a list of *(number of times it occurred, rating from 1 to 5)*. Top features include:

- **Goal Tracking** – Daily, weekly, and monthly tracking *[(16, 5)]*,
- **Web Support | One-Account Sync** – No task sharing, reasonable fee *[(15, 5)]*,
- **Automatic Task Scheduling** – Time slot organization by due dates *[(13, 5)]*,
- **Unlimited Completion Tasks** – For ongoing rewards *[(9, 5), (2, 3))*,
- **Repeating Event Tasks** – Flexible task assignment to events *[(8, 5), (1, 3)]*,
- **Routines** – Multiple projects after each other *[(5, 5), (4, 4), (3, 3)]*,
- **Task Templates** *[(7, 5)]*,
- **Rewards System Customization** *[(6, 5)]*,

- **Search Functionality** – Across events, projects *[(7, 3), (2, 5)]*.

The main aim for advancing TimeNoder2 is to enhance cross-platform compatibility. To accomplish this, introducing cross-platform functionalities and synchronization via Flutter is essential, with a focus on supporting Windows, macOS, and Linux, much like apps such as AnyType and AppFlowy. By implementing task-sharing features, it is possible to incorporate game-like elements for multiple users, which could potentially increase engagement in a domain known for high user dropout rates. Prioritizing synchronization through sharing might improve users' willingness to pay and drive growth. Furthermore, Flutter facilitates the development of a WearOS app for on-the-go tracking, which could be designed to function independently from a phone, unlike the Focus Todo smartwatch app. Other key tasks involve addressing bugs, enhancing localization, and improving widgets. Suggestions from Discord highlight new feature ideas, necessitating an assessment of the development requirements and the complexities they may introduce.

4.3 Detailed Requirements

User feedback highlights the need for cross-platform functionality and collaboration, necessitating robust account and team management features alongside an efficient synchronization system. This section specifies these requirements, laying the groundwork for subsequent design and implementation.

4.3.1 Account Management Requirements

Users require a secure and straightforward way to register and access the application. They must be able to:

- Sign up using an email address and password.
- Log in with their credentials securely.
- Recover a forgotten password via an email containing a reset link, redirecting them to the app or a web interface to set a new password.
- Begin using the application without the need for logging in.

4.3.2 Team Management Requirements

To support collaborative work, the application must enable team-based interactions. Key requirements include:

- Users can create a team, become its owner, and assign it a name.
- Team owners can generate invitation tokens to share with others, configurable as single-use or valid for a limited period (for example, 24 hours) to allow multiple users to join.
- Users can join a team by entering the token within the app.
- Team owners can remove members, and members can leave voluntarily.
- Owners can rename or delete the team, with deletion erasing all associated team data.

- Users can participate in multiple teams simultaneously.
- Teams have shared data accessible and editable by all members, while users can maintain private data not shared with a team.

4.3.3 Basic Synchronization Requirements

The synchronization system must efficiently manage a complex data model with numerous tables. It should:

- Handle a large number of database tables in a scalable way.
- Transition to a relational database to ensure data consistency and integrity.

4.3.4 Document Synchronization Requirements

Synchronizing documents presents unique challenges. The system must:

- Implement a merging strategy to combine edits from multiple users intelligently.
- Move beyond simplistic *"last-writer-wins"* conflict resolution approaches.
- Preserve the document's structure and intent.

4.3.5 Smartwatch Application Requirements

The smartwatch application aims to deliver a simplified yet functional extension of TimeNoder2 on WearOS, leveraging the existing Flutter-based mobile application. It must:

- Reuse the mobile app's database and repositories, adapting them for WearOS constraints.
- Adapt existing UI components to ensure usability on a small wearable screen.
- Enable users to view and track tasks associated with projects or specific categories (for example, pinned, today).
- Share timer status with the mobile app for a cohesive experience.
- Allow viewing and completing the current day's schedule, including recurring events.
- Allow completion of daily habits.
- Facilitate quick task creation directly from the wearable device.

4.4 Current TimeNoder2 Architecture

TimeNoder2 is presently an offline-only application, crafted using the Flutter framework. It adheres to an MVC architecture where Flutter is responsible for the view layer, Riverpod supervises the controller layer, and Isar DB underpins the model layer. The app was developed incrementally, incorporating features based on user responses. Employing Isar DB facilitated this process, as introducing new attributes or objects to existing entities typically sufficed to deploy new functionality. This strategy enables TimeNoder2 to provide a broad

spectrum of features: task management, scheduling, the Eisenhower matrix, a planning reward system, two-way synchronization with local calendars, highly customizable notifications, habit tracking, and detailed statistics.

Many of these features are deeply interconnected. This incremental development and the core features' extensive inter-connectivity resulted in a heavily denormalized database (Figure 4.1). This poses challenges for device synchronization, as it often requires a highly normalized database. Modifying the current database schema to a normalized format could substantially increase the number of tables, and adding multi-user synchronization would further complicate the schema. When database models are used directly in the view layer for ease of use, substantial alterations to these models can significantly affect the entire application.

A possible strategy is to align the restructured new database models, designed for single-user and multi-user synchronization, with the current models as closely as possible. This strategy helps to mitigate the potential side effects of the transition. Furthermore, it is noteworthy that employing Isar DB offers a significant advantage over traditional SQL databases. Isar DB allows most of the application to use synchronous code for accessing data. Combined with the database containing only a handful of object types, this greatly streamlines the data retrieval process. In contrast, local SQLite databases require asynchronous code due to their longer data retrieval times [69]. Additionally, in a normalized database, data is distributed across multiple tables, complicating data retrieval and storage compared to the current setup. All these factors suggest that the proposed modification would significantly affect the current architecture, which was not designed to accommodate such changes. This calls for a thorough exploration of design to determine a concrete solution.

Figure 4.1 shows a simplified current database schema for the TimeNoder2 application. The Event object is employed for specific occurrences within a time span, allowing a single Event to be linked with up to seven Tasks and up to five Projects or Sections. Events can also recur. Another significant entity is the Task, which can also recur. Tasks can optionally be associated with a Project or Section. The Project object directly contains Sections designated in the app using a `Project.id` combined with a `Section.id`, ensuring uniqueness within a Project. The attributes `eventId` and `calendarId` present on both Task and Event are crucial for synchronizing events within the local calendar and linking to a singular local event. The `createdInstances[]` attribute is used to denote created recurring Event instances, triggered either by creating a new event instance or by dynamic scheduling of notifications several days in advance. Events and tasks include notifications implemented with the same class, yet they function differently. Default notifications for tasks and events are stored in the settings, accommodating various options related to timers, notifications, etc. Additionally, Settings include FavouriteQuotes. Finally, two objects are connected to the rewards system: PointRecord, which records points earned for task completion, tracking, and planning, as well as expenses for purchased rewards.

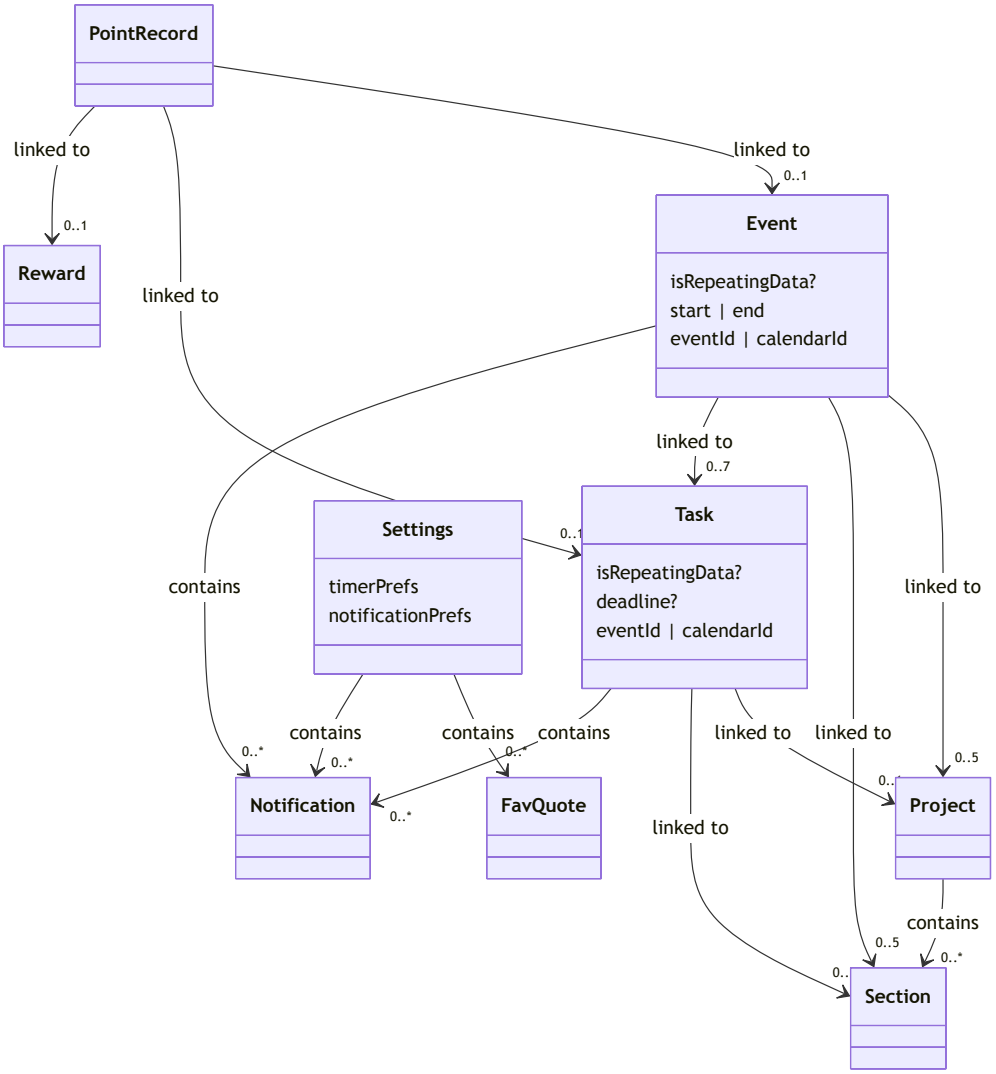


Figure 4.1: Diagram of a simplified current database schema for TimeNoder2.

Upon evaluating user needs, it becomes clear that there is significant demand for compatibility across different platforms (Section 4.2.1). Examining the existing architecture reveals ambiguity surrounding whether synchronization should target individual users or support multiple users. Moreover, the complexity involved in achieving synchronization poses a challenge, as it is just one aspect of building a cross-platform application. Consequently, the subsequent steps should meticulously focus on crafting synchronization mechanisms designed for single-user and multiple-user interactions. This design should also include the development of a smartwatch application to ensure its smooth integration within the overall system's synchronization framework.

Chapter 5

Design of the Solution

This chapter details the proposed database design for TimeNoder2, which shifts from a non-relational Isar DB to a relational database format to accommodate synchronization for single-user and multi-user environments. It explains the relational schema, includes Entity-Relationship Diagrams (ERDs) for single-user and multi-user contexts, and discusses document synchronization and smartwatch application design. The database design emphasizes a high degree of normalization to meet synchronization algorithms' demands and ensure efficient synchronization and a consistent database state.

5.1 Synchronzation Design

This section details the database schema design required for single-user and multi-user synchronization, including document synchronization. It also highlights crucial aspects related to implementation.

5.1.1 Basic Data Synchronization

This section will elaborate on the needed database design changes to allow one user to synchronize their data between multiple devices. This design will be later expanded to allow multi-user synchronization. It is vital that in the design, it is assumed that the synchronization will use the *"last-writer-wins"* conflict resolution strategy at the level of individual records. Significant effort is dedicated to ensuring that new models can be reasonably adapted to align with the current models, as these existing models are extensively employed throughout the application; thus, it is crucial to convert data to fit these models as much as possible.

The synchronization operates persistently in the background, either when the client possesses new data to sync or when notified by the server of changes that the client can access. Alternatively, polling can be used to check for updates regularly. Synchronization must also consider the internet connection status and the device's state. Essentially, it should initiate synchronization when the internet connection is re-established and when the application transitions from closed or in the background to active.

The synchronization must encompass all the tables listed below for single-user and multi-user use cases. Attempts to optimize this process are generally unnecessary, since the data is typically required in its entirety. For instance, displaying the tasks page necessitates access to all Tasks, Events, Projects, and Sections, as these are essential for presenting estimates and task details, effectively involving nearly the entire database (Figure 4.1).

This principle applies to other pages too. In the schemas presented below, the database undergoes normalization, but maintains a high level of connectivity that renders questions about which tables to synchronize and when unnecessary. All user-accessible data should remain consistently synchronized without any disruptions.

One-User Multi-Device Synchronization

As described in Section 4.4 on the TimeNoder2 architecture, the core models are intricately linked. To enable synchronization and enhance data integrity, the database must be restructured into a highly normalized format. The comprehensive ER diagram is divided into sub-diagrams and explained individually. Initially, one must recognize that this diagram presupposes a `users` table, which signifies distinct registered users. Within the subsequent schemas, each row's ownership is identified by `user_id`, linking it to the specific user associated with the record. The actual data are subsequently separated into multiple tables.

Data for projects and sections is stored in distinct tables (Figure 5.1). The `tasks` table includes two nullable references to the `sections` and `projects` tables. This design is implemented because a task can exist independently of any project. However, when a section reference is given, a project reference is also included to facilitate an easier transformation back to the original data model.

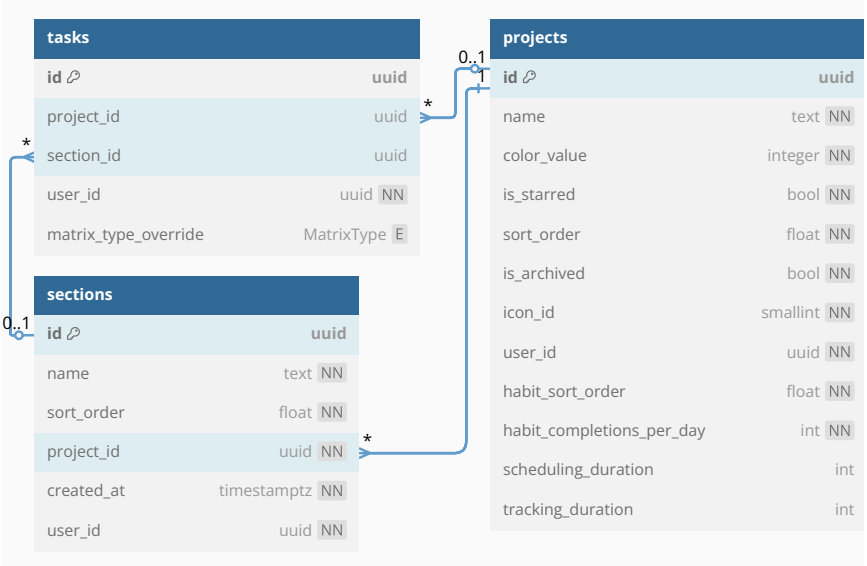


Figure 5.1: ER diagram for normalized storage of Projects and Sections.

The task data is organized into several tables (Figure 5.2): the `tasks` table, which holds fundamental task information, and the `repeating_task` table, which contains data specifying the task repetition rules and is consistently linked to a specific task. Upon task completion or skipping, if there is a subsequent task as per the recurrence rule, a new entry in the `tasks` table is generated, and `repeating_task.latest_instance` is updated to reference the upcoming task, thus maintaining its status as a repeating task. However, even when tasks cease to be repetitive, they remain connected to the repeating task by `repeating_task` record to facilitate tracking of the completion history. The `task_notifications` table is responsible for holding both pre-existing and user-generated task notification types, which can then be suggested, utilized by the user, and

associated with specific tasks via the `task_notifications_tasks` table. Additionally, `subtasks` are stored separately to enhance modification efficiency and allow for the addition and alteration of new subtasks from various devices. This configuration mitigates the risk of row-level *"last-writer-wins"* conflict resolution strategy override that might happen if both primary task attributes and task subtasks are modified simultaneously on various devices.

It is essential to clarify the difference in notifications for tasks and events. In TimeNoder2, tasks trigger notifications before their deadlines, while events generate alerts both before they start and before they end. Tasks usually deal with longer-term deadlines, thus requiring distinct notifications compared to events, which tend to demand less advanced notifications but can also notify before concluding. These distinctions mean they are structurally different and thus utilize separate tables.

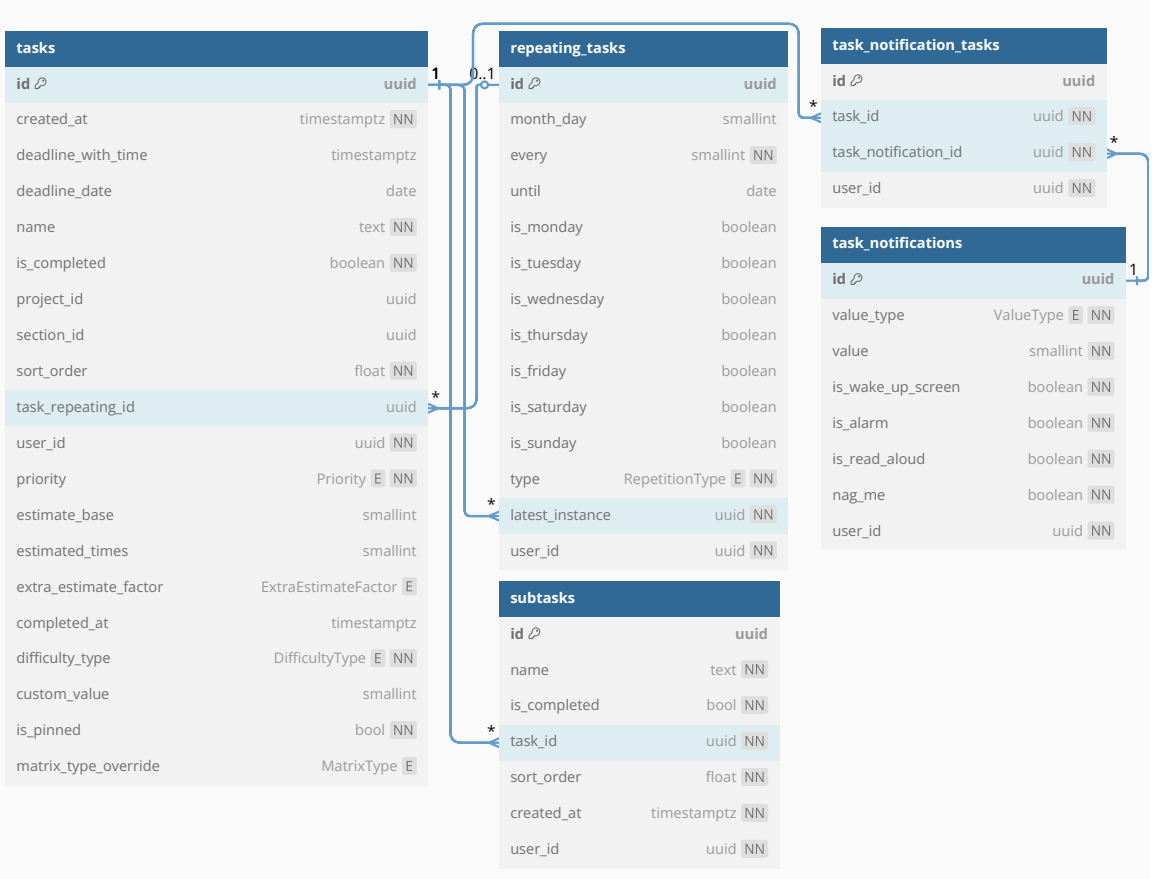


Figure 5.2: ER diagram for normalized storing of Tasks.

Data for events is arranged across various tables (Figure 5.3): the `events` table caters to both single and recurring events that have been accepted. The `repeating_events` table specifically manages repeating events by directly storing all essential details required to define them. Introduced is the `unified_events` table, which allows for the unique identification of events through an `id`, facilitating use in the user interface and assigning notifications, tasks, projects, and sections. The `event_notifications` table manages both predefined and user-generated notification types, which users can adopt and link to particular events via the `unified_events_notifications` table. The `uniqued_events_tasks` table is dedicated to assigning tasks to events, while the `unique_events_projects_sections` table is

tasked with associating projects and sections to events. These relationships are kept distinct since an event may only have a project assigned without any tasks, or it might include a task unrelated to a project, and hence, no projects. It is critical to remember that linking tasks, projects, and optional sections to events is interconnected; for instance, when Task A related to Project A is assigned, Project A should be automatically assigned to the event in the application. The table `missed_repeating_instances` is used for marking repeating events as missed.

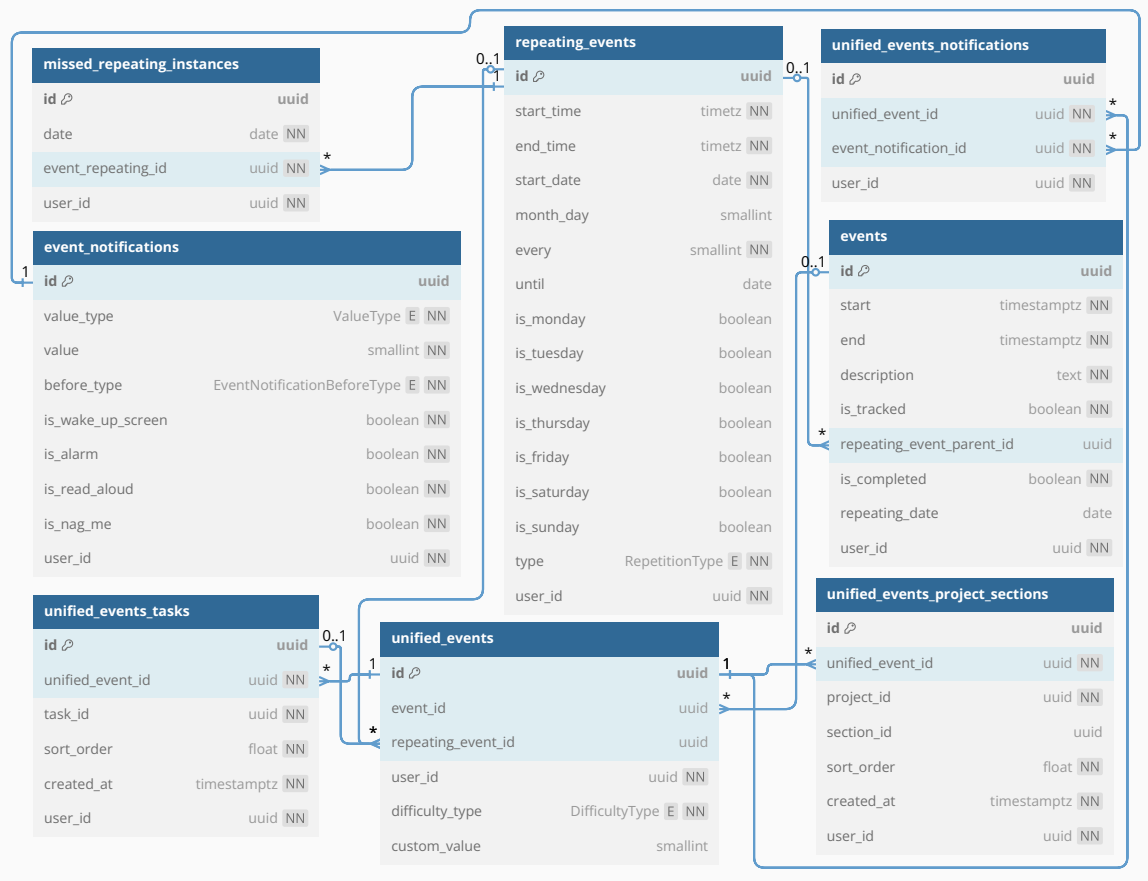


Figure 5.3: ER diagram for normalized storing of Events and their connection to Tasks, Projects, and Sections.

The logic for rewards has undergone significant changes from the original design to accommodate important user requests for better rewards customizability. This is accomplished through three tables associated with rewards (Figure 5.4). The `rewards` table comprises both predefined and user-generated rewards. The `reward_buys` table records reward purchases. The `reward_settings` table includes real-time calculation rules, enabling future and retrospective reward adjustments. Reward settings are stored in a JSON format, a deliberate compromise due to the uncertainty regarding user preferences for customizing the reward system, and to allow schema flexibility. Given the row *"last-writer-wins"* conflict resolution approach, this decision is reasonable as the settings pertain to a single user. The `settings` are stored in a similar manner and for comparable reasons, though they anticipate a greater degree of modification and change. The `quotes` table stores quotes

separately, which streamlines the database, enhances efficiency, and lays a better foundation for future quote-related features, such as sharing.

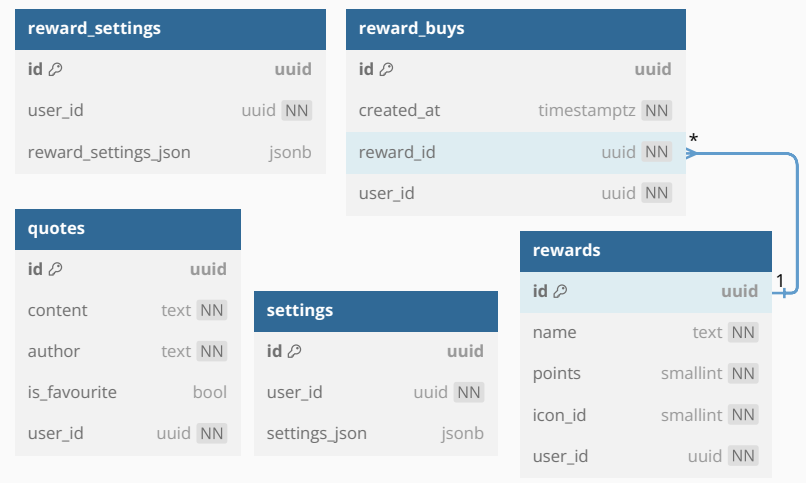


Figure 5.4: ER diagram for storing Settings and Rewards.

In a single-user context, it is important to recognize that the user has access only to the changes they have personally initiated. This implies that previously hidden information remains inaccessible to them. This feature is noteworthy as it allows timestamps to be effectively used for synchronization purposes. Nevertheless, this does not negate the value of more advanced strategies that do not require extra timestamp attributes. It simply suggests that in a single-user scenario, depending on timestamps is both practical and beneficial for maintaining control over the system instead of relying on complex third-party solutions. Consequently, the approach should focus on utilizing timestamps and avoiding third-party dependencies. The specific data schema and architecture remain closely integrated, thereby being left to the implementation's discretion.

Multi-User Multi-Device Synchronization

When the one-user database schema is extended to allow multi-user sharing, it consists of one fundamental change. For multi-user, the account management is expanded by a `accounts` table for personal and team accounts. All records were linked to a specific user using `user_id` in a one-user design. In multi-user design, the potentially shared elements are connected to a specific personal account or team account, using `account_id` (Figure 5.5). User-specific elements are still linked using `user_id` and enable users to define or override user-specific settings on a task, event, and project (Figures 5.6, 5.7, 5.8). For both tasks and events, there are also `*_assigns` tables that allow it to specify what should be seen by individual members of a team account in the UI. In cases of personal accounts, these tables would be empty.

The `accounts` are both personal and team-based (Figure 5.5). Personal accounts are always linked to one user using `primary_owner_user_id`, and relationships in `account_user` are expected to be empty. The `invitations` table stores invitations with all necessary details stemming from requirements.

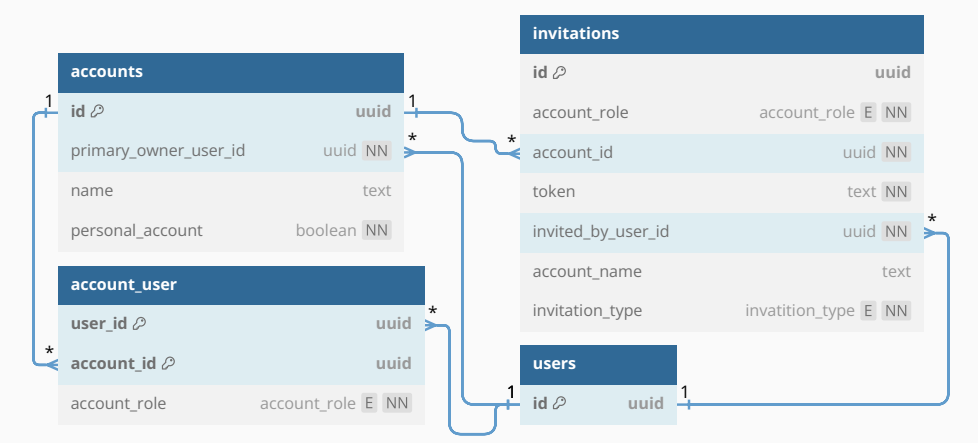


Figure 5.5: ER diagram for team management table.

In a team scenario, some of the previous attributes defined directly on the `tasks` table need to be moved to separate tables as they are specific to each user (Figure 5.6). On the client, all of these related tables: `task_difficulty`, `task_estimate_base_user_overrides`, `task_is_pinned` will be fetched together and combined with `tasks` table details, with some of these attributes missing, and combined to the same state as in a one-user scenario. The `task_assigns` allows for specifying who should see which tasks in a team.

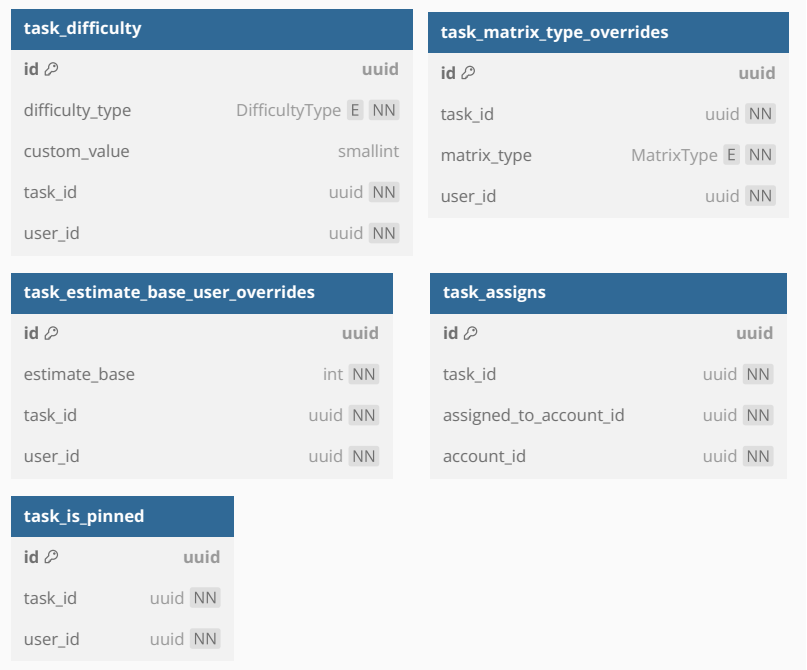


Figure 5.6: ER diagram of extra tables needed for defining Tasks in a multi-user environment.

The `unified_events_difficulty` enables users to define a reward for an event (Figure 5.7). The `unified_event_assigns` allows for specifying who should see which events in a team. Tables `project_is_starred`, `project_is_habit`, `project_trackings` and `project_is_archived` enable users to specify user specific project attributes (Figure 5.8).

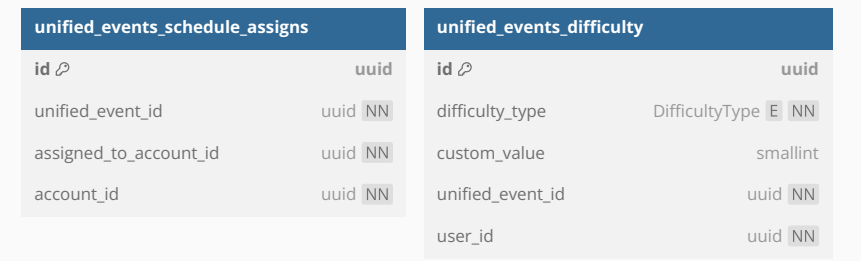


Figure 5.7: ER diagram of extra tables needed for defining Events in a multi-user environment.

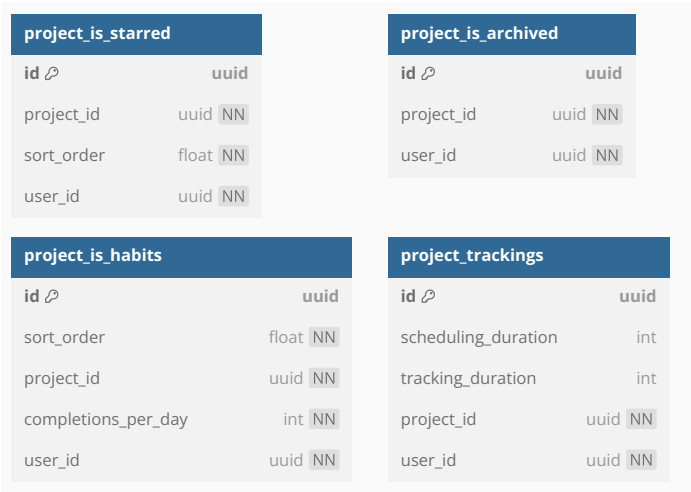


Figure 5.8: ER diagram of extra tables needed for defining Projects in a multi-user environment.

When data access is shared among multiple users, it presents synchronization difficulties absent in single-user contexts. This situation occurs when a user interacts with data generated before their most recent server sync, such as upon joining a team. This complicates dependence on timestamps. Algorithms relying on timestamps must be adapted to track when users access new data and notify them that a complete synchronization is necessary. In such situations, employing advanced solutions that leverage intermediaries instead of relying solely on timestamps becomes more practical. Consequently, the approach should focus on utilizing more sophisticated solutions. The specific data schema and architecture remain closely integrated, leaving it up for implementation.

5.1.2 Document Synchronzation

The previous design intentionally omitted the aspect of document sharing, which will now be addressed, building on the foundational synchronization for single and multiple user scenarios. Document sharing is presently managed by storing the document's state as markdown text, an effective solution for offline-only situations. However, for sharing document states across devices, it is crucial to enable partial updates synchronization. This approach conserves bandwidth and allows for integrating changes from various devices or users into a document.

Synchronization should utilize CRDTs (Section 2.2), which are particularly suited for such scenarios. The document will be transformed into a CRDT structure and represented in the database as a list of updates (Figure 5.9). Client-side, these updates can be merged to establish a CRDT structure that guarantees consistent results for all clients with identical updates. This structure can consequently be converted into a document state, which can be used to mirror both local and remote modifications within the text editor for the document.

The database implementation is nearly identical for both single-user and multi-user scenarios. A document serves as an element of task data, providing users the means to define a detailed description, and is thereby associated with it through `task_id`. In multi-user settings, the connection is made through `account_id`, which may refer to an individual or a team account. At the same time, in a single-user scenario, it relates to one particular user.

The document storage schema (Figure 5.9) is simple, consisting only of document updates stored in a `document_data` linked to a specific task. The `created_at` attribute is useful for filtering updates to apply to the CRDT structure.

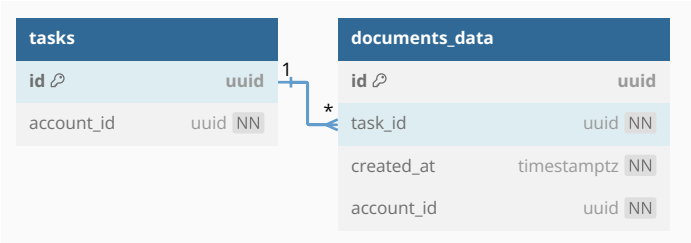


Figure 5.9: ER diagram for storing document data in a multi-user environment.

The particular approach for executing document synchronization relies on the available libraries and tools, and consequently must be tackled during the implementation phase.

5.2 Smartwatch Application Design

The smartwatch application should be developed using Flutter, enabling significant code reuse but restricting the app to WearOS. The application should provide a simplified version of the mobile app with full offline support, allowing features like the timer to function independently of the phone. Synchronization between WearOS smartwatches and associated devices should be established through the basic synchronization mechanism previously described. Since smartwatches can access the internet, this method is convenient and does not require extra implementation. The application should comprise six pages: `Tasks`, `Timer`, `Schedule`, `Habits`, `Create Task`, and `Settings`. Each page is designed for simplicity to ensure usability on the watch's small screen.

5.2.1 Tasks Page

The `Tasks page` is the initial screen, displaying today's tasks while providing access to tasks from other options and projects (Figure 5.10a). A button at the top shows the current selection, defaulting to *today*. Tapping this button navigates to the `Selection page` (Figure 5.10b), where users choose options such as *pinned*, *previous*, *today*, *tomorrow*, *future*, or projects. The `Tasks page` updates to display tasks relevant to the selected option. Below the button, a scrollable list of task tiles appears. Each tile includes a checkbox on the left,

the task name, additional details below, and a tracking button on the right. Tapping the tracking button initiates task tracking and navigates to the `Timer page`.

(a) Wireframe of `Tasks page` with selection button and list of task tiles.

(b) Wireframe of `Selection page`. Tapping a tile selects the option.

Figure 5.10: `Tasks page` and `Selection page`.

5.2.2 Timer Page

The `Timer page` appears when a user starts tracking a task or navigates directly to it. A description at the top indicates the timer's status, such as *"Work – Round 1 / 4"*. The current timer duration is shown below. If a tracking element is selected, it appears beneath the timer (Figure 5.11a). If no element is selected, an option to choose a task is displayed (Figure 5.11b). Tapping this option redirects to the `Tasks page` for task selection. Another option allows choosing a project, leading to the `Selection page` to pick a project or section. Below the tracking element, two buttons are present: a start or pause button, depending on the timer's state, and a skip button to bypass the current timer phase.

(a) Wireframe of `Timer page` with a tracked task, project, or section, highlighted in orange.

(b) Wireframe of `Timer page` without a tracking element. Selection options are highlighted in orange.

Figure 5.11: `Timer page` with and without a tracking element.

5.2.3 Habits and Schedule Page

The `Habits page` (Figure 5.12a) presents a list of habit tiles. Each tile starts with a habit label, followed by the habit's name and a compact completion button. This design allows users to track daily habit progress and mark completions directly on the watch.

The `Schedule page` (Figure 5.12b) offers a streamlined version of the mobile app's schedule, displaying the current day's events. Each event spans the full width, listing the time range, labels for projects, completion buttons for marking recurring events as done or missed, and the event's current progress.

(a) Wireframe of `Habits page`.

(b) Wireframe of `Schedule page`.

Figure 5.12: `Habits page` and `Schedule page`.

5.2.4 Create Task and Settings Page

The `Create Task page` (Figure 5.13a) includes a text input for the task name and an option to select a project. The deadline defaults to the current day. Users can add additional details on the phone later if needed.

The `Settings page` (Figure 5.13b) simplifies the configuration options of the mobile app, allowing users to customize the timer settings and watch notifications. Vibration alerts are given priority, as many users prefer them to audible sounds from the watch. Users can personalize the app using toggle switches or options that cycle through various choices. The settings are organized into distinct groups for easier navigation.

(a) Wireframe of `Create Task page`.

(b) Wireframe of `Settings page`.

Figure 5.13: `Create Task page` and `Settings page`.

Chapter 6

Implementation

The preceding chapter provided a comprehensive design overview for both data synchronization and the smartwatch application for TimeNoder2. Upon assessing the complexity involved in implementing single-user and multi-user synchronization, document synchronization, and the smartwatch extension, it became clear that such extensive modifications to the existing architecture go beyond the scope of this thesis. Converting TimeNoder2 to a fully normalized and synchronizable architecture would require a near-complete rewrite of the current application, affecting most source code files – many of which would need to be rewritten from scratch. This can be seen from ER diagrams presented in the previous chapter, where the original 6 models that contained data became 35 normalized tables, and this number can be even higher in production. The problem is that these models were used throughout the application, and the changes to the original models would break a lot of the original code.

Incorporating additional changes to the user interface to support desktop platforms and collaborative features, while maintaining usability, would further increase the complexity. Therefore, this thesis addresses core synchronization principles that are broadly applicable and beneficial to the application regardless of its final form. These principles also serve as a foundation for extending similar strategies to other systems in the future.

The first section focuses on the implementation of single-user and multi-user synchronization. These are demonstrated through prototype implementations that illustrate key foundational concepts, which can later be applied to TimeNoder2's data synchronization. These prototypes also cover document synchronization using CRDTs (Section 2.2), which will be elaborated on in the following sections. The second section is dedicated to the smartwatch application, addressing synchronization-related challenges and detailing the development process.

6.1 Synchronization Implementation

This section covers synchronization implementations for one-user and multi-user scenarios. Later, it adds an implementation for synchronization of documents. All these things are demonstrated in a separate demos that provide a good base for future TimeNoder2 development.

6.1.1 One-User Multi-Device Synchronization

One effective approach for basic data synchronization is inspired by the WatermelonDB protocol (Section 2.5.1). In single-user situations, the approach relies on the principle that these users cannot access previously concealed data, thus allowing the utilization of timestamps. It includes three key components: a local Drift SQL database[1], Supabase as a Backend-as-a-Service[2] (that can be easily replaced by a backend with access to SQL DB), and CRDT (Section 2.2) document synchronization. The local database is designed to save user modifications and send these changes to the server.

The specific process of synchronizing documents will be developed in a later section, but it is based on the principle described in Section 5.1.2. In summary, document changes will be saved in a database in the form of records and distributed across devices using the basic synchronization protocol. The architecture of this system is illustrated in Figure 6.1.

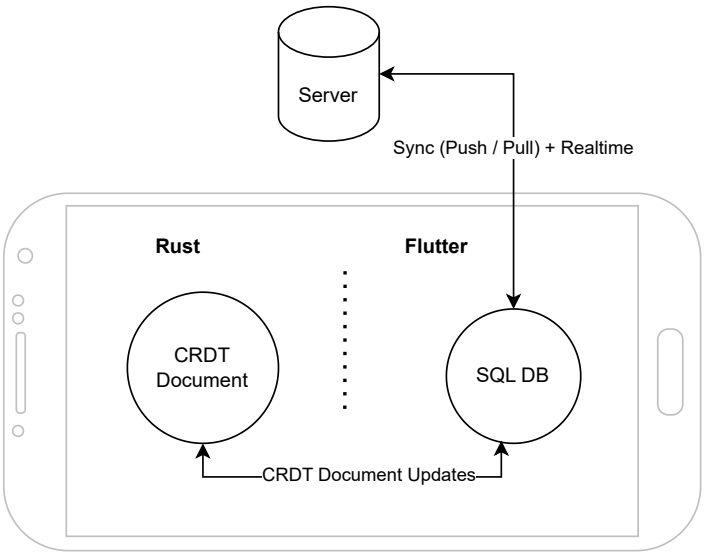


Figure 6.1: Functions *push* and *pull* refer to the WatermelonDB protocol functions. The real-time function corresponds to Supabase Realtime[3], which can be substituted with an equivalent service or with polling. The server can be anything that has access to a SQL database, and so it can run `push` and `pull` database functions.

Server Functions

The `pull` and `push` server operations adhere to the principles described in Section 2.5.1. They filter data accessible to the user through specific timestamp attributes and the `user_id`. Rather than relying solely on user_id checks, Row-Level-Security (RLS)[4] rules can be utilized to automatically restrict data access within database functions. The server uses key timestamp attributes to manage this: `server_created_at` tracks when a record is created on the server, `server_updated_at` logs when a record gets updated, and `deleted_at` marks when a record is deleted.

[1] https://pub.dev/packages/drifts
[2] https://supabase.com/
[4] https://www.postgresql.org/docs/current/ddl-rowsecurity.html

The core aspect of the push and pull functions lies in their generic nature. Thus, these functions operate effectively by focusing on the presence of additional timestamp attributes in synchronized tables. Moreover, they utilize the *changes* map, which organizes tables according to their dependencies. In the context of a todo list application, it is essential to structure the collection in the following order: Project, Task, and Event (which includes tasks). This dependency order ensures that all related elements are processed correctly.

Client-Side Synchronization

On the client, the changes are tracked using `created_at`, `updated_at`, and `deleted_at` timestamps as stems from the WatermelonDB protocol (Section 2.5.1). The important part of the implementation was that by a little bit of annotation on the used Drift DB database schema, it was possible to automate a big part of the protocol using code-generation (Listing 5).

```
// 1. Annotate Drift tables with @customSync
@customSync
class Project {
  // 2. Specify serverTableName
  serverTableName => "public.project"

  // 3. Add isRemote attribute
  isRemote => withDefault(false);
}

// 4. Specify classes in the order of their dependencies
@SyncManager(classes: [Project, Task, Docup])
class SyncClass {}
```

Listing 5: An illustration of pseudocode demonstrating the addition of essential annotations to the Drift DB schema for code generation.

After the code generation is run, it generates a `SyncManager` class with methods that are utilized in the main synchronization function using the WatermelonDB protocol:

- `syncedTables()` – Returns a list of table names in order of their dependencies.
- `sync(changes)` – Accepts `changes` object and applies the changes from the server to the local DB.
- `getChanges(lastPulledAt, db)` – Returns all changes that the server does not know about since `lastPulledAt` time.

One-User Multi-Device Demo

Demo[5] for one-user synchronization uses the previously defined synchronization mechanism. On the server, it defined tables with additional attributes (Figure 6.2) on which the algorithm relies. It allows for creating projects and tasks where each task contains a document, showcasing both one-user synchronization and synchronization of documents.

[5] https://habitmaster-e52e9.web.app/

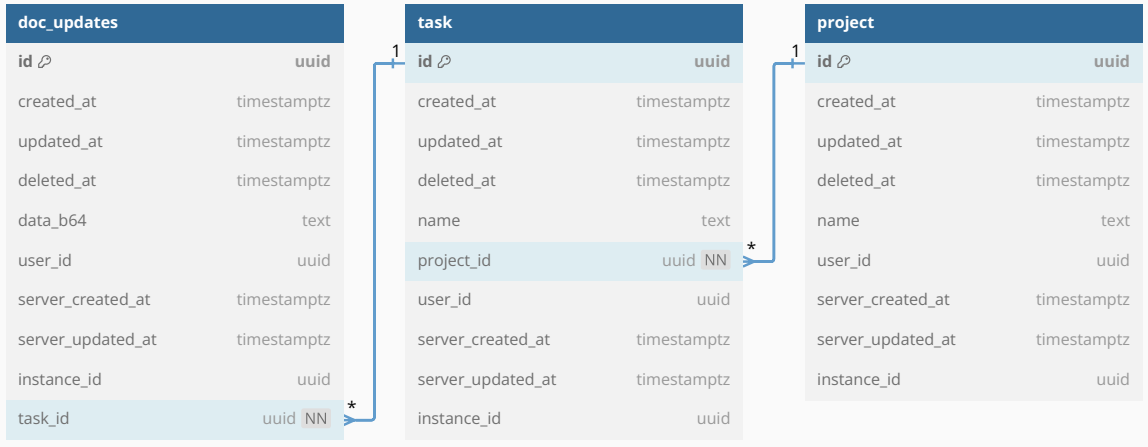


Figure 6.2: Server database schema utilized on the Supabase server for the one-user synchronization demo. On the client, it uses the same schema except the `instance_id`, `server_created_at`, and `server_updated_at` attributes. The `instance_id` filters out real-time updates that the current client is causing and limits the number of synchronization calls.

6.1.2 Multi-User Multi-Device Synchronization

As discussed in Section 5.1.1 regarding multi-user synchronization design, adding synchronization for multiple users introduces complexity due to scenarios where one user might access previously restricted resources. It must also address situations where multiple users attempt simultaneous insertions onto the server, potentially leading to conflicts, particularly with numerous changes occurring rapidly.

To tackle these challenges, the PowerSync synchronization framework, as detailed in Section 2.5, is employed, offering a robust synchronization solution. PowerSync's significant benefits include its independence from timestamps and its high-quality Flutter integration[6]. With the capability to self-host PowerSync[7] alongside Supabase using solutions like Coolify[8], it emerges as a feasible option for TimeNoder2 in the context of multi-user synchronization.

The architecture is similar to the one-user synchronization architecture, but instead of *push* and *pull* functions, PowerSync middleware monitors database changes using sync rules and forwards updates to clients (Figure 6.3). These sync rules enable the generation of client schemas for Flutter. From these schemas, a Drift ORM schema can be created, which provides an ORM layer on top of the Flutter PowerSync database[9].

[6] https://github.com/powersync-ja/powersync.dart/tree/main
[7] https://docs.powersync.com/self-hosting/getting-started
[8] https://docs.powersync.com/integration-guides/coolify
[9] https://pub.dev/packages/drift_sqlite_async

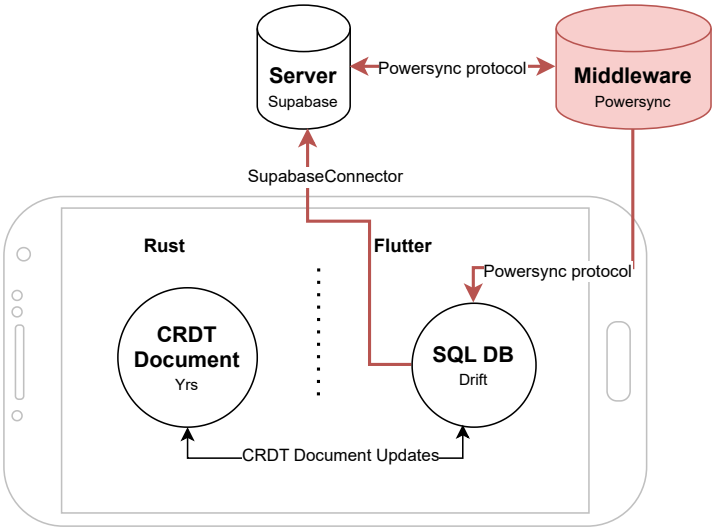


Figure 6.3: PowerSync Synchronization Architecture.

Client Implementation

Drift DB can be used on the client side without any modifications. Behind the scenes, it interfaces with PowerSync, logging changes in an update queue through three main operations: the `PUT` operation, derived from `INSERT` statements, creates a new row with values for all non-null columns; the `PATCH` operation, driven by `UPDATE` statements, modifies an existing row by applying changes identified by row `id` and updated column values; and the `DELETE` operation, executed via `DELETE` statements, eliminates a row specified by its row `id` [52]. These updates must subsequently be propagated to the server.

The task is accomplished by extending `PowerSyncBackendConnector` and implementing three essential functions: *fetchCredentials()* is responsible for retrieving authentication tokens, *invalidateCredentials()* manages authentication errors, and *uploadData()* transmits previously queued updates to the server as CRUD operations. For backend operations with Supabase, an existing template that processes CRUD updates sequentially is available. However, performance and consistency enhancements can be achieved using database functions[10] if necessary, as it allows sending a whole transaction at once.

Multi-User Multi-Device Demo

A demo was created to showcase the core multi-user, multi-device synchronization to lay a solid foundation for future work. The demo consists of sharing documents as personal documents as well as documents that are part of a team. It builds upon the multi-user multi-device synchronization schema defined in Section 5.1.1. The server uses Supabase with the Basejump template[11] for team management that largely resembles the previous design. An additional *user_account_access* table enables team members to see other members' account details (Figure 6.4).

[10] https://supabase.com/docs/guides/database/functions
[11] https://usebasejump.com/

This table is automatically maintained by triggers and represents a view of the *account_user* table. This is necessary because PowerSync only supports simple queries in synchronization rules. Without this, it would not be possible to synchronize the personal accounts of users connected through a team. An essential feature is that it permits users to handle documents without being logged in, while allowing synchronization with additional devices at a later stage.

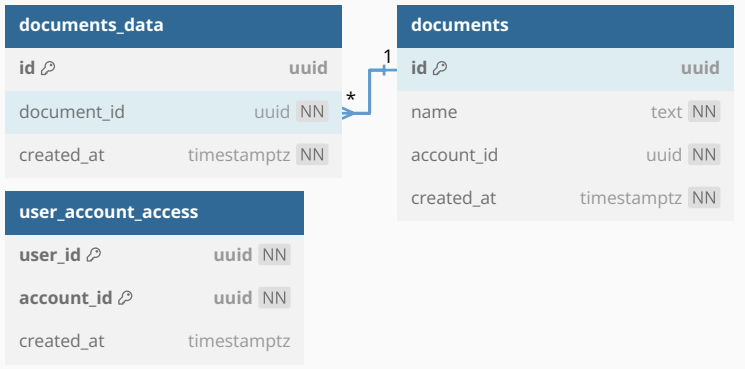


Figure 6.4: A demo ER diagram illustrates document sharing for both a single user and several users.

6.1.3 Synchronization of Documents

Basic synchronization techniques are not sufficient for document synchronization, as updates from various devices need to be integrated. The conventional `"lastwriter-wins"` technique falls short because it replaces changes rather than integrates them. The objective is to consolidate edits from multiple devices so that all devices eventually display the same final version of the document according to previously established criteria (Section 5.1.2). The proposed solution will be grounded in the earlier design (Section 5.1.2) and transform document changes into a series of updates that can be subsequently synchronized using base synchronization methods. This should clarify how document synchronization operates in the previously mentioned demonstrations for single-user (Section 6.1.1) and multi-user (Section 6.1.2) scenarios.

After evaluating a range of options, the **yrs**[12] library developed in Rust emerges as the most effective solution. This library features CRDTs (Section 2.2), making it well-suited for synchronizing modifications within a distributed context. Appflowy[13] follows a comparable methodology by utilizing `appflowy_editor`. To streamline implementation, the application should transition from *super_editor* to `appflowy_editor` due to superior support and maintenance, and a more appropriate API for synchronization processes. It also has markdown import capability that eases the transition. Moreover, *super_editor* faces financial constraints[14] and has an unstable API, further justifying the change.

Structure of the `appflowy_editor` Document Format

To synchronize documents using CRDTs effectively, it is essential to comprehend the underlying structure. The `appflowy_editor` document format is arranged as a hierarchical

[12] https://github.com/y-crdt/y-crdt
[13] https://github.com/appflowy-io/appflowy
[14] https://github.com/superlistapp/super_editor/issues/2270

JSON object that depicts a rich text document. At the highest level, the document includes a root block featuring a `type` field designated as `"page"` and a `children` array. Within this array, each item is a block representing a distinct type of content, such as a paragraph, heading, list, image, or other block component. Every block is comprised of these elements:

- `type` – Defines the category of the block (such as `"paragraph"` or `"heading"`).
- `data` – Contains properties specific to the block type. This generally includes a `delta` array for most text-related blocks, utilizing the Quill Delta format[15]. The `delta` array comprises a sequence of insert operations representing text segments, which may feature formatting attributes such as `bold`, `italic`, and `underline`. Other stored properties include, for example, `align` for text alignment or `level` for heading depth.
- `children` – An array consisting of block identifiers representing the sequence of a block's children.

In the `CRDT Document`, the design will be depicted as a `Blocks` map of type YMap, where attributes are saved as strings, except for the delta attribute, which will be stored as YText, a special type for text operations. Contrary to some expectations, the final structure will solely employ `Blocks` map and not utilize a YMap of YArray to manage the order of children blocks within each parent block. A detailed explanation for this decision is provided in the following synchronization workflow.

This workflow consists of two parts: reflecting user-created changes in the `CRDT Document`, and reacting to changes from outside the current device by merging them into the current `CRDT Document` and reflecting these changes in the text editor.

Reflecting User-Created Changes in the CRDT Document

Reflecting user-created changes in the CRDT document is handled in 4 stages (Figure 6.5).

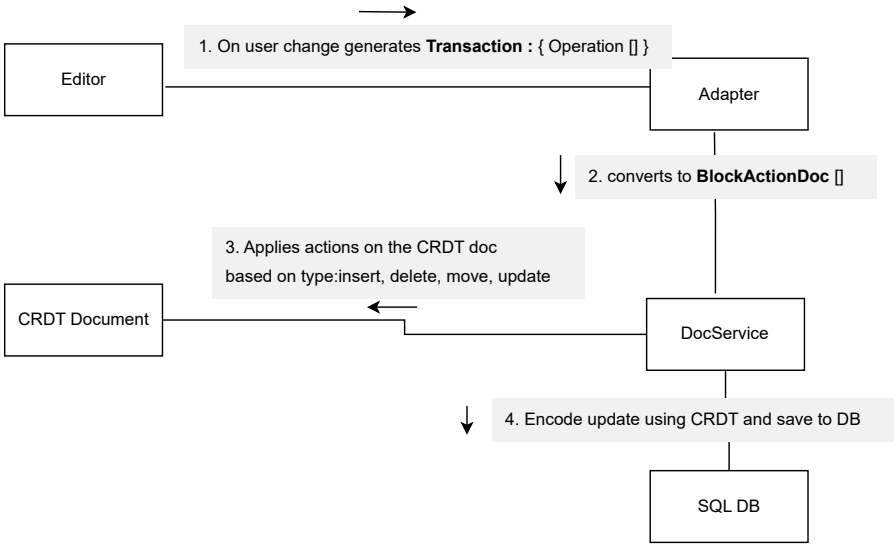


Figure 6.5: Reflecting user-created changes in the `CRDT Document`.

[15] https://docs.yjs.dev/api/delta-format

1. **Transaction Generation**
   The process begins when the editor generates a `Transaction` capturing document changes. Each `Transaction` consists of a list of `Operations` that can be applied to the UI or sent to the backend for storage and transformation.

2. **Adapting Transaction**
   The `Transaction` is transformed into a list of `BlockActionDoc`, where each contains data from a `Operation`, but also includes extra information needed to reflect changes in the `CRDT Document`. Besides the actual node data and type, it includes:

   - `prevId` – Points to the previous block on the same level or null if there is no previous block.
   - `nextId` – Points to the next block on the same level or null if there is no next block.
   - `parentId` – Points to the parent block or null if it is a root block.
   - Extra data attributes added to the data attribute:
     - `deviceId` – Corresponds to the `id` assigned to the device for one synchronization session.
     - `timestamp` – Corresponds to the local datetime when the block was last updated.

   These attributes are reflected in the `CRDT Document` and used when extracting data from the `CRDT Document`.

3. **Reflecting Adapted Transaction in a CRDT Document**
   Based on the `BlockActionDoc`, it reflects changes in the `CRDT Document` in the following ways:

   - Updates the block data and type, together with additional attributes `deviceId` and `timestamp`.
   - Updates the `parentId` chain.
   - Updates the `prevId` chain.

4. **Computing difference**
   After changes are applied to the `CRDT Document` in the current transaction, a binary-encoded CRDT update is computed (Listing 6). This update is then returned to the Flutter side.

```
before_state = txn.before_state();
update = txn.encode_diff(before_state);
```

Listing 6: By comparing a transaction's starting and ending states, `yrs` generates a binary encoded CRDT update corresponding to applied changes.

5. **Saving Into DB**
   On the Flutter side, this update is saved as a binary or base64 encoded binary update (as tools such as PowerSync[16] do not support binary data) and then synchronized between devices using the basic synchronization protocol.

Reflecting Remote Changes Within the Editor

This workflow involves receiving updates from the database, originating either from the current device or others, and integrating them into a `CRDT Document`. This document is then compared with the existing editor state, allowing the differences to be applied to the text editor, thereby synchronizing their states. The procedure is divided into six steps (Figure 6.6):

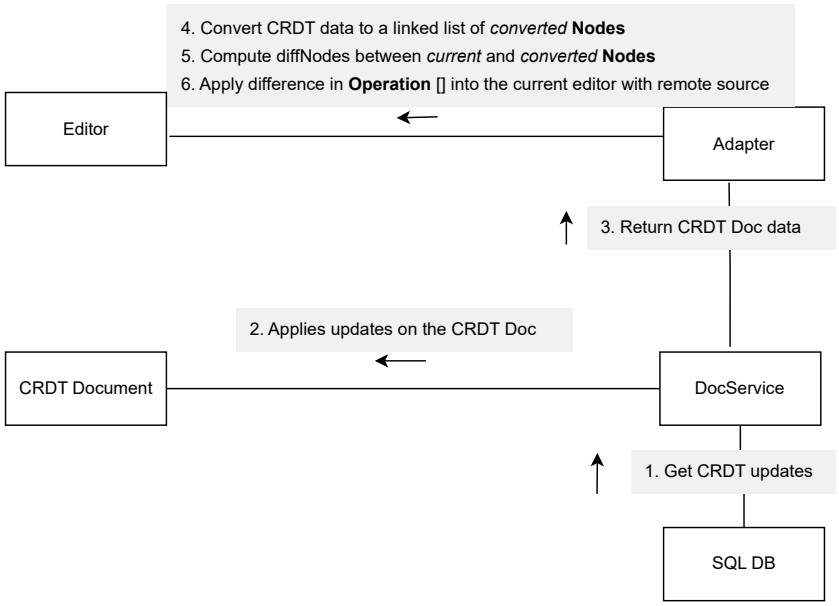


Figure 6.6: How updates from other devices are reflected in the editor.

1. **Sending CRDT document updates to DocumentService**
2. **Applying updates to the current CRDT document**
3. **Returning the CRDT document to Flutter for comparison**

   The resulting document is sorted before it is returned. The sorting algorithm is implemented in Rust and uses multiple criteria:

   - First groups blocks by `parentId`.
   - Within each parent group, sorts blocks by `deviceId`.
   - Follows `prevId` chains to preserve intended ordering.
   - Uses timestamps to resolve conflicts when multiple blocks have the same `prevId`.

[16] https://docs.powersync.com/usage/sync-rules/types

- Ensures devices' contributions stay grouped.

This sorting algorithm is crucial for maintaining user expectations when merging concurrent document changes. An important consideration: the Appflowy implementation does not rely on `prevId` and `nextId` pointers like this implementation does. It uses an extra Yrs map called `children_map` where each parent has a list of children's `ids` stored in a Yrs array. While more straightforward, the problem appears when multiple devices edit offline and then synchronize their changes. Consider this example:

User A (Offline)

```
- 111 - User A
- 222 - User A
- 333 - User A
```

User B (Offline)

```
- 111 - User B
- 222 - User B
- 333 - User B
```

When both devices synchronize, a naive CRDT merge with blocks and children maps might produce an interleaved result that is not user-friendly:

```
- 111 - User A
- 111 - User B
- 222 - User A
- 333 - User A
```

Using the system described above, the plugin sorts blocks to produce an expected merge, where changes made on one device are grouped together:

```
- 111 - User A
- 222 - User A
- 333 - User A
- 111 - User B
- 222 - User B
- 333 - User B
```

Alternatively, depending on the sorting criteria, the result could be:

```
- 111 - User B
- 222 - User B
- 333 - User B
- 111 - User A
- 222 - User A
- 333 - User A
```

While it is important not to undermine Appflowy, which has functional collaborative software, the architecture of their system remains a mystery in this thesis.

4. **Converting the CRDT document into an Editor Structure**
   The document is converted into a linked list of `Nodes`, a structure that `appflowy_editor` understands.

5. **Computing the Difference**
   The current editor's `Nodes` linked list and the previously transformed linked list undergo a diffing algorithm. The original *diff* method provided by `appflowy_editor` produced incorrect results with improper ordering of `DeleteOperations`. Instead, a custom *diffWithDocument* method compares the two documents and returns a list of `Operations` that `appflowy_editor` can process.

6. **Applying operations to the current editor**
   This step reflects remote changes in the editor using *apply(diff, remote: true)* API that `apflowy_editor provides`.

Currently, remote updates are applied with a longer debounce delay (*syncDebounceDelay*, default 1500ms) because all combined changes are used for comparison. This includes updates stored in the database and updates returned after applying changes to the `CRDT Document` but not yet saved. Doing this too often would result in junk in the UI. Also, local updates are not saved directly to the DB as it would create a large number of records. Instead, local changes are batched together with *updatesBatcherDebounceDuration* (default 500ms), merged using `yrs`, and then saved to the database. Both debounce attributes can be customized.

Another challenge that this implementation needed to address was the potential for entering an invalid state while navigating within the editor. This issue was reported, but no response has been received yet[17]. Ultimately, the solution is integrated into the synchronization protocol, which validates the current state of the editor. When changes like this occur, the protocol ensures that they are corrected.

Document Synchronization Plugin API

The earlier described algorithm for document synchronization has been converted into a library called `appflowy_editor_sync_plugin`[18], which is now available for other developers to utilize. The `appflowy_editor` GitHub repository[19] features a demonstration of how this plugin can be used.

[17] https://github.com/AppFlowy-IO/appflowy-editor/issues/1083
[18] https://github.com/Musta-Pollo/appflowy_editor_sync_plugin
[19] https://github.com/AppFlowy-IO/appflowy-editor

The synchronization system provides an API that developers can adopt with minimal setup. The *EditorStateSyncWrapper.init()* function generates the `EditorState` object, which is used by `appflowy_editor` to manage state. Developers only need to define three methods on the `SyncAttributes` class for the document synchronization to work (Listing 7).

```
editorState = EditorStateSyncWrapper(
  syncAttributes: SyncAttributes(
    /// Get initial document state
    getInitialUpdates: () {},

    /// Stream of all saved updates for the document
    getUpdatesStream: () => db.documentUpdatesStream

    /// Save local update to the database
    saveUpdate: (update) {},
  ),
).init();
```

Listing 7: Developer API pseudocode for document synchronization using created `appflowy_editor_sync_plugin`[20].

In summary, by using the `yrs` library for CRDTs and the API provided by `appflowy_editor`, changes from multiple devices can be reliably merged. This ensures that all devices eventually display the same document state with user-expected ordering, overcoming the limitations of traditional synchronization methods that rely on a *"last-writer-wins"* conflict resolution strategy.

6.2 Smartwatch Application Implementation

For the smartwatch application implementation, it is essential to understand that the original design (Section 3.2) cannot be fulfilled as the synchronization between devices was not added to TimeNoder2. Because of this, the synchronization relies on the native API between the phone and its companion smartwatch application. The WearOS app communicates exclusively with Android phones, as stems from prior research (Section 3.2). Communication between the smartwatch and phone utilizes the `watch_connectivity` plugin for Flutter, which provides an API for data exchange.

This section describes the implementation of a WearOS smartwatch application developed using Flutter. The application works alongside a companion mobile app on a connected smartphone, with both sharing the same bundle identifier for seamless communication. Using Flutter enables significant code reuse across platforms, particularly for database logic, timer services, and some user interface components.

6.2.1 Communication Protocol

Communication between devices uses the `watch_connectivity` plugin. Because message delivery between watches and phones can be unreliable and bandwidth is limited, a custom

protocol was developed to ensure data consistency. This protocol includes five message types:

1. **Heart_beat**: Sent periodically to verify that the other device is connected and working.
2. **Sync_suggest**: Sent when local changes are detected to prompt the other device to start synchronization.
3. **Sync_init(lastCompletedAt)**: Request changes that occurred after the specified `lastCompletedAt` timestamp.
4. **Sync_data(inserted_data, deleted_data)**: Contains actual data changes, including new and deleted items.
5. **Sync_completed(startSynchronization)**: Notifies that synchronization has finished, including changes up to the `startSynchronization` timestamp.

To handle unreliable message delivery, the protocol includes a confirmation system implemented through the `ConfirmableWatchMessageHandler` class. Each message waits for acknowledgment before proceeding, which simplifies synchronization logic and reduces communication overhead. After initial synchronization, later synchronizations typically need only a single message exchange, as the `sync_data` and `sync_completed` messages can be combined when possible to save bandwidth.

6.2.2 Local Database and Synchronization

A critical aspect of this implementation is the integration of a local database directly onto the smartwatch. Unlike other solutions such as Focus To-Do that rely on a constant connection to a smartphone, this strategy ensures that all modifications are recorded locally. This functionality safeguards against data loss from connectivity problems between a watch and a phone. Once the connection is restored, synchronization is triggered, allowing both devices to refresh each other with any updates, thus improving reliability and supporting offline app usage. Synchronization is executed using the protocol described previously, utilizing the existing database architecture with Isar. In addition to the data types being synchronized between the smartwatch and mobile app, two new attributes were introduced:

- a timestamp attribute, `updated_at`, indicating when the record was last modified,
- a boolean attribute, `synced`, indicating if this record got to the current device by synchronzation.

These new attributes facilitate the identification of records requiring synchronization. Furthermore, an additional record type for deleted entries was introduced to ensure that deletions are also synchronized.

6.2.3 App Design and Navigation

Designing for WearOS required careful consideration of platform-specific constraints. A significant challenge was that the default leftward swipe gesture exits the application, which made conventional navigation patterns like page views (used in apps like Focus To-Do) impossible. To solve this problem, a global menu bar button was integrated into the interface. This button is designed to be noticeable but not intrusive, and when activated, it displays an overlay with navigation options to other pages.

6.2.4 App Features

The smartwatch app consists of several core pages, each representing a streamlined version of the mobile app's features. These pages were developed as outlined in the previous chapter, including a `Create Task page`. The `Create Task page` functioned properly on a physical device used during development; however, text input issues arose on the emulator and during the Google Play review process, leading to its removal. Apart from this, all the remaining pages have been created. These pages emphasize simplicity and user-friendliness, tailored to the restricted watch display. To maintain consistency and reduce development time, fundamental interface elements like buttons and labels are adapted from the mobile application.

Chapter 7

Testing

This chapter presents the testing of the implemented synchronization and the smartwatch application. Synchronization was tested in two previously presented demos, one of which demonstrates the basic synchronization mechanism using the WatermelonDB protocol along with document synchronization (Section 6.1.1), and another demo focuses on team management and document sharing, while being directly integrated into TimeNoder2 (Section 6.1.2).

7.1 One-User Synchronization Demo

This demo tests both WatermelonDB synchronization and document synchronization. The test involved five different testing scenarios, all completed successfully. The basic synchronization focuses on sharing tasks and projects between devices. Document synchronization testing focuses on concurrent editing, insertions, text operations, deletions, move operations, and offline editing. One testing recording scenario is available on the demo GitHub repository[1]. One of the testing scenarios is described in detail in the Appendix A.1 to allow replication.

Note that WatermelonDB synchronization is more suited for native platforms. On the Web, it is used primarily to demonstrate synchronization of basic data and documents without requiring installation. Currently, each tab needs to create a new local database file and synchronize changes to it. This is due to a known issue with Drift when two tabs attempt to write to the same database file simultaneously[2].

During testing, some things were identified that are not considered issues themselves, but rather stem from the used strategy. It is when two users edit the same line, and the debounce durations are higher in the code. Users can then write new lines of text without seeing each other's changes. If both edit the same line and create children under the same line, users could view the behavior after the merge as unexpected. The recommendation is to use, in addition to the current method, the method provided by Appflowy for real-time collaboration that consists of directly sharing transactions produced by the `appflowy_editor` in real-time.

[1] https://github.com/Musta-Pollo/custom_supabase_drift_doc_sync/tree/main
[2] https://github.com/simolus3/sqlite3.dart/issues/240

7.2 Multi-User Synchronization Demo

The testing consisted of a single testing scenario, as document synchronization was already thoroughly tested in the previous section, and PowerSync is a widely adopted commercial product that does not require extensive testing here. The primary goal of this demo is to showcase the fundamental synchronization principles used when creating a collaborative cross-platform application. Testing was performed on a physical Android phone and an iOS emulator. The testing scenario is available in the Appendix A.2.

A few features regarding team management suggested in the analysis phase are not shown here, as they follow the same principles as the current implementation. The current implementation uses the Basejump template for team management, which supports creating teams, inviting members, etc. These Basejump-managed tables are synchronized using PowerSync. For example, removing a team member would involve calling the database function: *remove_account_member*[3]. This change would then be synchronized across devices via PowerSync.

For deleting a team account, there is no built-in API in Basejump template. In such cases, a custom database function must be defined in Supabase[4], or the deletion must be handled locally and synced through the `SupabaseConnector`. With the current `SupabaseConnector` implementation and proper foreign key relationships (with cascading deletes), deleting a team account – assuming the user has permissions as enforced by PostgreSQL RLS – would also remove related documents and document updates. All changes would then be synchronized to the related devices.

7.3 Smartwatch Application Testing

The smartwatch testing includes a testing scenario to verify functionality and synchronization, and user testing to evaluate experience and satisfaction.

7.3.1 Scenario Focused on Functionality and Synchronization

The process begins with initializing the mobile device, which will be less detailed than previously, as it is not the core focus of this thesis. The sole aim here is to produce test data for demonstration purposes. Following this, the smartwatch application testing examines the synchronization of tasks, events, projects, and the timer status. It comprehensively evaluates the timer's functionality concerning data persistence, adherence to WearOS guidelines for required functionalities[5], and reliability. Finally, it verifies that users can perform basic operations in the smartwatch application. The assessment was carried out using a physical Android smartphone and a connected physical WearOS smartwatch. The detailed testing scenario for verifying smartwatch functionality and synchronization is in the Appendix A.3.

7.3.2 User Experience Testing

The user experience testing of the smartwatch app helps evaluate the application's user experience. It is important to note that TimeNoder2 was based on user reviews, often marked as an application that offers a lot of techniques and tools. From this naturally

[3] https://usebasejump.com/docs/team-members#remove-team-member
[4] https://github.com/usebasejump/basejump/issues/87
[5] https://developer.android.com/distribute/best-practices/launch/distribute-wear

steps complexity, which is confirmed by the user reviews on Google Play, but also by the messages that users share on the app's Discord server.

As of the day this is written, the application has not successfully passed the Google Play review to be published in the production track, and so the testing cannot be done with actual users. As the watch application cannot be shared differently than by directly installing it using `adb`[6], it was required to share a personal phone and a smartwatch to carry out the tests on them. Because of this, the testing focuses mainly on understandability and whether users can achieve various tasks.

At the beginning, the users were in approximately 5 minutes introduced to TimeNoder2. After that, the application on the phone and the smartwatch application were reset to the default state, and the actual testing began. The testing consisted of:

- **User-data Preparation** – Preparing data in the mobile application, crucial for following smartwatch application testing.
- **Smartwatch Application Testing** – Testing focused on navigating and executing basic actions in the smartwatch application.

User-data Preparation

1. Open the mobile app, and create projects called "Math", "Fitness".
2. Create two tasks: "homework math" for "Math" project with a deadline tomorrow, and for today, "evening run" for "Fitness" project.
3. Make a "Fitness" project a habit.
4. Go to the `Schedule page` and create an event from 17:00 to 17:30 with the "evening run" task assigned.

Smartwatch Application Testing

1. Open the smartwatch application and view your daily plan.
2. Complete the "evening run" task.
3. Verify that your daily habit of "Fitness" is completed.
4. Look at what tasks you have for tomorrow.
5. Track the "homework math" task for 1 minute. Close the application on the watch.
6. After 1 minute, stop the task tracking.
7. Change the application settings so that the timer is not using 24-hour notation, but rather 12-hour notation.
8. Verify that it did what was desired on the `Schedule page`.

[6] https://developer.android.com/tools/adb

The user testing involved three participants: one with minimal prior experience with TimeNoder2 and the other two completely new to it. This lack of previous experience greatly affected the results of the tests. The main challenge was effectively demonstrating the application to testers, highlighting key features but omitting many details. Testers certainly required more than five minutes to thoroughly understand TimeNoder2, which would significantly enhance the application's relevance on a smartwatch. After completing the introduction and setting up default tasks, participants transitioned to the smartwatch application. All testers observed longer loading times, but then all quickly located the navigation menu, denoted by the hamburger menu icon, and accessed the `Schedule page`. With more experience, this process would have felt intuitive. When asked to complete the "evening run" task, two participants did so directly on the `Schedule page`. The third navigated back to the `Tasks page` to finish the task. Familiarity with the mobile app would have helped all testers recognize the `Schedule page`'s layout and complete the task there.

Upon reaching the `Tasks page`, which they quickly adapted to, testers easily found the button at the top. Though somewhat uncertain if they made the correct choice, they selected "tomorrow" and viewed the task list for the following day. Since this test was pre-structured, they were unsure if they were on the task list for tomorrow. This issue would be resolved with their actual data. For tracking, participants easily identified the tracking button on the task tile. However, all three users were unsure if they had started the timer, and the third paused it because of confusion. This stems from the timer operating as a Foreground Service, which requires a moment to activate. Phones, with superior performance, exhibit this delay less noticeably. This behavior should be carefully monitored and addressed as necessary.

After closing the app and waiting, all users hesitated on the empty home screen before scrolling to the notification list, as they were not using their personal smartwatches. Upon locating the timer notification, two participants first paused the timer and, after encouragement, ended it. The third participant directly ended the timer. When asked to adjust settings, users swiftly tapped the navigation button and selected settings. They easily enabled the first option, which was the disabled 12-hour mode.

In conclusion, the smartwatch application's navigation and user elements were generally easy to understand. It was notably more accessible to grasp than the mobile version. The most significant issues identified were the initial confusion with the timer, which remained in the starting state for a significant time before transitioning to the running state, and the longer loading time. An additional familiarity with the mobile application would resolve other uncertainties and hesitations.

Chapter 8

Conclusion

This thesis emerged from the increasing popularity of the author's application, TimeNoder2[1], as many users requested a cross-platform version and were willing to pay for it. This interest led naturally to the undertaking of transforming TimeNoder2 into a cross-platform application that supports collaboration, a project that aligned well with the master's thesis requirements. The journey led to exploring what it truly entails to make TimeNoder2 both cross-platform and collaborative, addressing the challenges within. Ultimately, it became apparent that TimeNoder2 was not equipped for this transition. Hence, the thesis concentrated on fulfilling its objectives by delving into core synchronization issues, such as data exchange between multiple users, sharing collaborative documents, and designing a smartwatch integration.

Initially, existing synchronization solutions were examined for both basic relational data and collaborative documents synchronization. Conflict-free replicated data types (CRDTs) were thoroughly investigated because they provide a dependable framework for simultaneous collaborative editing by multiple users. A substantial part of the thesis focused on enhancing TimeNoder2 to support smartwatches running on WearOS. Given that Flutter supports deployment to WearOS, much of the existing codebase was utilized, easing the development process. The research further delved into the reasons behind smartwatch purchases, primary consumer interests, and relevant scientific insights regarding their advantages. Subsequently, the design and functionality of smartwatch applications were reviewed to enhance the adaptation of TimeNoder2 for this platform.

An in-depth examination of the current TimeNoder2 application was carried out, incorporating user feedback to understand user expectations and necessities better. Subsequently, two synchronization methods were introduced. The initial solution relied on the WatermelonDB protocol but was modified to utilize the Drift database, a widely respected SQLite Flutter database. While it currently operates assuming a PostgreSQL backend database, it can be modified for other relational databases. The setup employs client-side code generation along with universal logic on the server side, facilitating ease of extension – developers merely need to set up tables with the appropriate structure on both local and server databases, with the rest being managed automatically. This approach was acknowledged by the Drift creator, who listed it among notable synchronization strategies[2]. It is also user-friendly for customization and adjustment. This solution was presented in a functional demo that included document synchronization[3].

[1] https://play.google.com/store/apps/details?id=com.janzimola.goal_venture2&hl=en&gl=US
[2] https://drift.simonbinder.eu/examples/server_sync/#manual
[3] https://habitmaster-e52e9.web.app/

The second approach utilized the PowerSync synchronization engine alongside the Basejump template for Supabase. This demonstration illustrated a potential vision for TimeNoder2 as a fully cross-platform collaborative application, encompassing basic data and document synchronization features. Furthermore, a smartwatch edition of TimeNoder2 was developed. According to user feedback, the app was user-friendly, with minor concerns about loading times and timer initiation. It presented a streamlined version of TimeNoder2, adhering to WearOS design principles, while enabling users to manage tasks while on the move. One significant drawback was the lack of capability to generate tasks directly on the watch. The interface allowing this feature was omitted as the text input field, though operational during development, resulted in crashes, which prevented the app from successfully completing Google Play's review process. Despite a few setbacks and challenges during publishing, the app was ultimately published successfully.

Additional emphasis was placed on the collaborative sharing of documents. The approach incorporates CRDTs for their advantageous characteristics. Even though these CRDT map and array structures are specifically tailored for document editing, issues were encountered during prototyping, particularly when merging CRDT arrays after offline edits, as the results were not as expected. Ultimately, CRDTs serve to synchronize document states among connected devices. Furthermore, a custom sorting algorithm, complemented by additional attributes, maintains the user's intent. This functionality was packed into a plugin that is designed for the `appflowy_editor` library, which was selected as the best text editor option for collaborative editing in this thesis. The plugin was acknowledged by the `appflowy_editor` library, which also valued its contribution, and currently features a demo of this synchronization plugin in its showcase.

In conclusion, this thesis presents various insightful strategies for creating cross-platform and collaborative applications with Flutter, including those for smartwatches. These insights remain significant regardless of whether TimeNoder2 eventually evolves into a collaborative tool. While synchronization constitutes just a piece of the puzzle, this research lays a robust foundation for future development endeavors.

Bibliography

[1] Abuwarda, Z.; Mostafa, K.; Oetomo, A.; Hegazy, T. and Morita, P. Wearable devices: Cross benefits from healthcare to construction. *Automation in Construction*. Elsevier, 2022, vol. 142, p. 104501.

[2] Ahn, C. R.; Lee, S.; Sun, C.; Jebelli, H.; Yang, K. et al. Wearable sensing technology applications in construction safety and health. *Journal of Construction Engineering and Management*. American Society of Civil Engineers, 2019, vol. 145, no. 11, p. 03119007.

[3] Android Developers. *Conserve power and battery* https://developer.android.com/training/wearables/apps/power. 2024. Accessed: 2024-06-17.

[4] Android Developers. *Wear OS Design and Development Guide* https://developer.android.com/design/ui/wear/guides/. 2024. Accessed: 2024-12-15.

[5] Anytype Team. *Any-Sync Protocol Overview*. 2024. Available at: https://tech.anytype.io/any-sync/overview. Accessed: 2024-11-25.

[6] Anytype Team. *Data Storage and Deletion*. 2024. Available at: https://doc.anytype.io/anytype-docs/data-and-security/data-storage-and-deletion. Accessed: 2024-11-25.

[7] AppFlowy-IO. *AppFlowy-Collab*. 2024. Available at: https://github.com/AppFlowy-IO/AppFlowy-Collab. Accessed: 2024-11-25.

[8] AppFlowy Team. *Database Monitoring With Realtim*. 2023. Available at: https://docs.appflowy.io/docs/guides/appflowy/self-hosting-appflowy-using-supabase#database-monitoring-with-realtime. Accessed: 2024-11-25.

[9] AppFlowy Team. *Database*. 2024. Available at: https://docs.appflowy.io/docs/documentation/software-contributions/architecture/backend/database. Accessed: 2024-11-25.

[10] AppFlowy Team. *How we built Appflowy with Flutter and Rust*. November 2024. Available at: https://appflowy.io/blog/tech-design-flutter-rust. Accessed: 2024-11-25.

[11] Apple Developer. *WWDC23: Design and build apps for watchOS 10* https://www.youtube.com/watch?v=BPJZ6A_brSw. 2024. Presented by Jennifer Patton (Apple Design Team) and Matthew Koonce (SwiftUI Team for watchOS), Published on May 3, 2024.

[12] Bai, Y.; Tompkins, C.; Gell, N.; Dione, D.; Zhang, T. et al. Comprehensive comparison of Apple Watch and Fitbit monitors in a free-living setting. *PLoS One*. Public Library of Science San Francisco, CA USA, 2021, vol. 16, no. 5, p. e0251975.

[13] Barman, L.; Dumur, A.; Pyrgelis, A. and Hubaux, J.-P. Every byte matters: Traffic analysis of bluetooth wearable devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*. ACM New York, NY, USA, 2021, vol. 5, no. 2, p. 1–45.

[14] Blascheck, T.; Besançon, L.; Bezerianos, A.; Lee, B. and Isenberg, P. Glanceable visualization: Studies of data comparison performance on smartwatches. *IEEE transactions on visualization and computer graphics*. IEEE, 2018, vol. 25, no. 1, p. 630–640.

[15] Burnham, J. and Dellera, S. *Introduction to UX Design on Wear OS* YouTube video. 22. May 2023. Available at: https://www.youtube.com/watch?v=fZpVlbzlvuY. Presenters: Josef Burnham, UX Designer; Sofia Dellera, UX Designer.

[16] Carroll, M. and Pateman, M. *Introduction to UX Research & Product Inclusion on Wear OS* YouTube video. 22. May 2023. Available at: https://youtu.be/puGn72d86qw?si=MYjDImW_idEHJHjf&t=44. Presenters: Mallory Carroll, UX Researcher; Matthew Pateman, UX Researcher.

[17] Casaccia, S.; Revel, G. M.; Scalise, L.; Cucchieri, G. and Rossi, L. Smartwatches selection: market analysis and metrological characterization on the measurement of number of steps. In: IEEE. *2021 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*. 2021, p. 1–5.

[18] Cheatham, S. W.; Stull, K. R.; Fantigrassi, M. and Motel, I. The efficacy of wearable activity tracking technology as part of a weight loss program: a systematic review. *J Sports Med Phys Fitness*, 2018, vol. 58, no. 4, p. 534–548.

[19] Chen, X.; Chen, W.; Liu, K.; Chen, C. and Li, L. A comparative study of smartphone and smartwatch apps. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, p. 1484–1493.

[20] Cook, J. *YouTube Video: Unknown Title*. February 2023. Available at: https://www.youtube.com/watch?v=kUlt27KmHDc. Accessed: 2024-11-25.

[21] Counterpoint Technology Market Research. *Global Smartwatch Market Forecast: 2024 and Beyond* https://www.counterpointresearch.com/insights/global-smartwatch-market-2024/. 2024. Accessed: 2024-12-11.

[22] Davidson, S. B.; Garcia Molina, H. and Skeen, D. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR)*. ACM New York, NY, USA, 1985, vol. 17, no. 3, p. 341–370.

[23] Dudjak, M. and Martinović, G. An API-first methodology for designing a microservice-based Backend as a Service platform. *Information Technology and Control*, 2020, vol. 49, no. 2, p. 206–223.

[24] Faiz, M. and Shanker, U. Data synchronization in distributed client-server applications. In: IEEE. *2016 IEEE International Conference on Engineering and Technology (ICETECH)*. 2016, p. 611–616.

[25] Flutter Community. *Watch_connectivity: Flutter plugin for communication between watch and phone*. 2025. Available at: https://pub.dev/packages/watch_connectivity. Accessed: 2025-05-02.

[26] Foster, K. R. and Torous, J. The opportunity and obstacles for smartwatches and wearable sensors. *IEEE pulse*. IEEE, 2019, vol. 10, no. 1, p. 22–25.

[27] Garcia, B.; Chu, S. L.; Nam, B. and Banigan, C. Wearables for learning: examining the smartwatch as a tool for situated science reflection. In: *Proceedings of the 2018 CHI conference on human factors in computing systems*. 2018, p. 1–13.

[28] Gomes, V. B.; Kleppmann, M.; Mulligan, D. P. and Beresford, A. R. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages*. ACM New York, NY, USA, 2017, vol. 1, OOPSLA, p. 1–28.

[29] Google Developers. *Send and receive messages on Wear*. 2025. Available at: https://developer.android.com/training/wearables/data/messages. Accessed: 2025-05-02.

[30] Google Developers. *Sync Data*. 2025. Available at: https://developer.android.com/training/wearables/data/data-layer. Accessed: 2025-05-02.

[31] Ioannidis, D. C.; Kapasouri, E. M. and Vassiliou, V. S. *Wearable devices: monitoring the future?* Oxford University Press, 2019.

[32] Jiang, T.; Yang, J.; Yu, C. and Sang, Y. A clickstream data analysis of the differences between visiting behaviors of desktop and mobile users. *Data and Information Management*. Elsevier, 2018, vol. 2, no. 3, p. 130–140.

[33] Kaashoek, F.; Li, K.; Marsh, B. and Tauber, J. A. Storage alternatives for mobile computers. In: *Symposium on Operating Systems Design and Implementation (Monterey CA) pages*. P. 25–39.

[34] Kass, T.; Coffey, J. and Kass, S. Bridging the Gap Between Desktop and Mobile Devices. In: Springer. *HCI International 2020–Late Breaking Posters: 22nd International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings, Part I 22*. 2020, p. 134–141.

[35] Kawell Jr, L.; Beckhardt, S.; Halvorsen, T.; Ozzie, R. and Greif, I. Replicated document management in a group communication system. In: *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*. 1988, p. 395.

[36] Keng, J. C. J.; Jiang, L.; Balan, R. K.; Lee, Y. and Misra, A. Profiling power utilization behaviours of smartwatch applications. In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion*. 2016, p. 96–96.

[37] Khononov, V. *Learning Domain-Driven Design*. „ O'Reilly Media, Inc.“, 2021.

[38] Khurana, R.; Banovic, N. and Lyons, K. In only 3 minutes: perceived exertion limits of smartwatch use. In: *Proceedings of the 2018 ACM International Symposium on Wearable Computers*. 2018, p. 208–211.

[39] King, C. E. and Sarrafzadeh, M. A survey of smartwatches in remote health monitoring. *Journal of healthcare informatics research*. Springer, 2018, vol. 2, p. 1–24.

[40] Kleppmann, M. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. „ O'Reilly Media, Inc.“, 2017.

[41] Kleppmann, M. *CRDTs and the Quest for Distributed Consistency* YouTube video. InfoQ, 2. October 2018. Available at: https://youtu.be/B5NULPSiOGw. Accessed: 2024-11-22.

[42] Krug, S. *Don't make me think!: a common sense approach to Web usability*. Pearson Education India, 2000.

[43] Kulkarni, S. S.; Demirbas, M.; Madappa, D.; Avva, B. and Leone, M. Logical physical clocks. In: Springer. *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*. 2014, p. 17–32.

[44] Long, J. *DotJS 2019 - James Long - CRDTs for Mortals* https://youtu.be/DEcwa68f-jY?si=DLqadz4cx6LFJL1v. 2019. Conference video. Accessed: 2024-11-17.

[45] Lui, G. Y.; Loughnane, D.; Polley, C.; Jayarathna, T. and Breen, P. P. The apple watch for monitoring mental health–related physiological symptoms: Literature review. *JMIR Mental Health*. JMIR Publications Inc., Toronto, Canada, 2022, vol. 9, no. 9, p. e37354.

[46] Mahar, K. *The Best To-Do List App*. 2024. Available at: https://www.nytimes.com/wirecutter/reviews/best-to-do-list-app/. Accessed: 2024-11-25.

[47] Martin, S.; Ahmed Nacer, M. and Urso, P. Abstract unordered and ordered trees CRDT. *ArXiv preprint arXiv:1201.1784*, 2012.

[48] Miret, L. P. Consistency models in modern distributed systems. an approach to eventual consistency. *Master. MA thesis. Universitat Politecnica de Valencia, Spain*, 2014.

[49] Müller, B. *Offline-First React Native Apps with Expo, WatermelonDB, and Supabase*. 2023. Available at: https://supabase.com/blog/react-native-offline-first-watermelon-db. Accessed: 2024-11-25.

[50] Pätz, C.; Michaelis, A.; Markel, F.; Löffelbein, F.; Dähnert, I. et al. Accuracy of the Apple Watch oxygen saturation measurement in adults and children with congenital heart disease. *Pediatric Cardiology*. Springer, 2023, vol. 44, no. 2, p. 333–343.

[51] Perez, M. V.; Mahaffey, K. W.; Hedlin, H.; Rumsfeld, J. S.; Garcia, A. et al. Large-scale assessment of a smartwatch to identify atrial fibrillation. *New England Journal of Medicine*. Mass Medical Soc, 2019, vol. 381, no. 20, p. 1909–1917.

[52] PowerSync. *Writing Client Changes*. Available at: https://docs.powersync.com/installation/app-backend-setup/writing-client-changes. Accessed: 2025-05-10.

[53] PowerSync Team. *Postgres and Yjs CRDT: Collaborative Text Editing Using PowerSync*. 2024. Available at: https://www.powersync.com/blog/postgres-and-yjs-crdt-collaborative-text-editing-using-powersync. Accessed: 2024-11-25.

[54] PowerSync Team. *PowerSync Sync Rules*. 2024. Available at: https://github.com/powersync-ja/powersync-service/blob/main/packages/sync-rules/README.md. Accessed: 2024-11-25.

[55] Shadiev, R.; Hwang, W.-Y. and Liu, T.-Y. Investigating the effectiveness of a learning activity supported by a mobile multimedia learning system to enhance autonomous EFL learning in authentic contexts. *Educational Technology Research and Development*. Springer, 2018, vol. 66, p. 893–912.

[56] Shapiro, M.; Preguiça, N.; Baquero, C. and Zawirski, M. Conflict-free replicated data types. In: Springer. *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. 2011, p. 386–400.

[57] Van Steen, M. and Tanenbaum, A. S. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[58] Velmovitsky, P. E.; Alencar, P.; Leatherdale, S. T.; Cowan, D. and Morita, P. P. Using apple watch ECG data for heart rate variability monitoring and stress prediction: A pilot study. *Frontiers in Digital Health*. Frontiers Media SA, 2022, vol. 4, p. 1058826.

[59] Vilarinho, T.; Farshchian, B.; Bajer, D. G.; Dahl, O. H.; Egge, I. et al. A combined smartphone and smartwatch fall detection system. In: IEEE. *2015 IEEE international conference on computer and information technology; ubiquitous computing and communications; dependable, autonomic and secure computing; pervasive intelligence and computing*. 2015, p. 1443–1448.

[60] Visuri, A.; Sarsenbayeva, Z.; Van Berkel, N.; Goncalves, J.; Rawassizadeh, R. et al. Quantifying sources and types of smartwatch usage sessions. In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 2017, p. 3569–3581.

[61] Vogels, W. Eventually Consistent: Building reliable distributed systems at a worldwide scale demands trade-offs? between consistency and availability. *Queue*. ACM New York, NY, USA, 2008, vol. 6, no. 6, p. 14–19.

[62] WatermelonDB Developers. *Backend*. 2024. Available at: https://watermelondb.dev/docs/Sync/Backend. Accessed: 2024-11-25.

[63] WatermelonDB Developers. *Backend*. 2024. Available at: https://watermelondb.dev/docs/CRUD. Accessed: 2024-11-25.

[64] WatermelonDB Developers. *Synchronization*. 2024. Available at: https://watermelondb.dev/docs/Sync/Intro. Accessed: 2024-11-25.

[65] Yasmin, A.; Mahmud, T.; Debnath, M. and Ngu, A. H. An Empirical Study on AI-Powered Edge Computing Architectures for Real-Time IoT Applications. In: IEEE. *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2024, p. 1422–1431.

[66] Zaslavsky, A. and Tari, Z. Mobile computing: Overview and current status. *Journal of Research and Practice in Information Technology*, 1998, vol. 30, no. 2, p. 42–52.

[67] Zhou, X.; Zaslavsky, A.; Rasheed, A. and Price, R. Efficient object-oriented query optimisation in mobile computing environment. *Journal of Research and Practice in Information Technology*, 1998, vol. 30, no. 2, p. 65–76.

[68] Zhu, X.; Guo, Y. E.; Nikravesh, A.; Qian, F. and Mao, Z. M. Understanding the networking performance of wear OS. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*. ACM New York, NY, USA, 2019, vol. 3, no. 1, p. 1–25.

[69] Zimola, J. *Smart Task Planner: Inteligentní Plánovač Úkolů*. Brno, Czech Republic, 2023. Bachelor's Thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Hynek, P.

Appendix A

Testing of Synchronization and Smartwatch Application

For document testing, the markdown syntax supported by `appflowy_editor` is used. For example, when the text says: "Tab1 writes a heading 'Heading' on the first line", it means that the raw input was "# Heading". For a bullet-list item, it would be written as "- [list item content]". For special types, such as a checkbox, type "/" and choose an appropriate element from the context menu. This simplification is important for document-related testing and is used to make the steps easier to follow.

A.1 One-User Synchronization Scenario

Testing scenario of basic synchronization using WatermelonDB protocol, including synchronization of documents. It consists of:

- **Testing Preparation** – That prepares initial data that is used to test that, before stored data is loaded on login.
- **Verifying Pre-Created Data** – Verification that data stored in the previous phase is loaded on login.
- **Editing in Two Tabs** – Simultaneous editing in two tabs, performing operations such as inserting, updating, deleting, moving, and offline editing.

A.1.1 Testing Preparation

1. Open a new browser window and go to the demo website: Demo[1].
2. Log in using the demo credentials provided on the login page.
3. Create a new project called "Before Demo", open the created project tab, and create a new task: "Before demo task".
4. Open the document editor for the task: "demo task". Enter a heading: "# Heading" and add three bullet points: "point 1", "point 2", "point 3". Then wait about 10 seconds, and then close the browser window.

[1] https://habitmaster-e52e9.web.app

A.1.2 Verifying Pre-Created Data

1. Open a new browser window and go to the demo website: Demo[2].
2. Log in using the provided demo credentials.
3. Open the demo website in a second tab and place both tabs side by side. In both tabs, you should see the "Before Testing Project".
4. Open "Before Testing Project" in both tabs — the task "Before Testing task" should appear in each.
5. Open the "Before Testing task" and then its document in both tabs. The document editors should show the previously entered heading and bullet points. Go back to the home page in both tabs.

A.1.3 Editing in Two Tabs

1. In tab 1, create a project named "Demo Thesis Project". After synchronization, the project should be visible in both tabs.
2. In tab 2, open the project page for "Demo Thesis Project" and create a task named "Demo task".
3. In tab 1, open the same project – the task "Demo task" should now be visible in both tabs. Open "Demo task" in tab 1, and then open the task's document editor.
4. Do the same in tab 2 – at this point, both editors should show an empty document.
5. In tab 1, add a heading "Heading" on the first line by writing "Heading". After synchronization, the heading should appear in both tabs.
6. In tab 2, modify the heading by appending "a" to make it "Headinga". After synchronization, both tabs should show "Headinga". Cursor positions should remain preserved (tab 1's cursor after "g", tab 2's cursor after "a").
7. In tab 1, delete the last three letters and replace them with "xxx" → should now be "Headixxx". Synchronization should reflect this change everywhere.
8. In tab 2, select "xxx" and make it italic via context menu. After synchronization, both tabs should show "xxx" in italic.
9. In tab 1, add a bullet point below the heading and name it: "point 1".
10. In tab 2, add two more bullet points: "point2", "point3", then insert a horizontal separator by typing three times "-", and finally add another heading: "New Heading". After synchronization, it should show the same structure in both tabs.
11. In tab 1, delete the "Headingxxx" line and the three bullet points below it. Both tabs should now show an empty heading line, the horizontal separator, and the "New Heading".
12. In tab 1, create a bullet point under "New Heading" called "point with tasks".

[2] https://habitmaster-e52e9.web.app

13. In tab 2, create a task under “point with tasks” by turning the bullet into a checkbox named “task 1”, then add “task 1.1” nested under it, and “task 1.1.1” nested under that.

14. Still in tab 2, add a regular bullet point below: “point without tasks”.

15. Move “task 1” to be under “point with tasks”. After synchronization, both editors should reflect this structure correctly.

16. Under “task 1”, add another task: “task 2”. Now the structure should be task 1 → task 2 → task 1.1.

17. In tab 1, delete “task 2”. The remaining structure: task 1 → task 1.1 → task 1.1.1 should be visible in both tabs.

18. Move “task 1” above “point with tasks”. After synchronization, “point with tasks” will have only task 1.1 nested under it, task 1 above.

19. Lock the device and come back after 15 minutes.

20. In tab 1, insert a new heading “First line heading” at the top.

21. In tab 2, delete everything. After synchronization, both editors should show an empty document.

22. In tab 1, write a heading: “Now testing offline sync”. Wait 10 seconds and turn off the internet.

23. Put both cursors at the end of “Now testing offline sync”. Create a new line in each tab.

24. In tab 1, write a heading “Heading Tab 1” with two bullet points “point tab 1 1”, “point tab 1 2”, followed by a horizontal separator.

25. In tab 2, do the same – but replace “tab 1” with “tab 2”.

26. In tab 2, under the separator, add “Exchanging sides Tab 2” with checkboxes: “checkbox tab 2 1”, “checkbox tab 2 2”.

27. In tab 1, do the same as above but with “tab 1” instead. Turn the internet back on. After synchronization, both editors should show the same structure: “Heading Tab 1”, “Exchanging Tab 1”, “Heading Tab 2”, “Exchanging Tab 2”. Sometimes, tab 2 content may come first, but the important part is that changes from tab 1 and tab 2 are not mixed.

28. In tab 1, delete: “tab 1 1”, “Heading Tab 2”, “checkbox tab 2 2”.

29. In tab 2, delete the heading: “Now testing offline sync” and leave a blank line at the top.

30. In tab 1, remove the initial character from “Heading Tab 1” or “Heading Tab 2”, based on the content of the second line. Subsequently, remove the entire first line.

31. In tab 2, insert a new heading above “Heading Tab 1”: “First spot heading”. Swap bullet point “point tab 1.1” with “point tab 1.2”. Eventually, both editors should show the same final state. Close tab 1.
32. In tab 2, add a bullet point “tab 2 2” at the end of the document.
33. Open tab 3, go to the same task editor, and confirm that the documents match.

A.2 Multi-User Synchronization Scenario

Testing scenario of team synchronization using the PowerSync framework, including synchronization of documents. It consists of:

- **Offline-Only Mode and Registration** – Verification that offline-only mode is supported and that users can create accounts.
- **Team Management and Document Editing** – Verification that users can do basic team management and collaborate on shared documents.

A.2.1 Offline-Only Mode and Registration

1. Delete application data on both Android and iOS.
2. Open the app on both devices.
3. Walk through the introduction screens without creating demo data.
4. Allow notification permissions to simplify testing.
5. On device A (Android phone), open the drawer and create a new personal document by clicking: “Add personal document”. Name it “device A personal document”.
6. On device B (iOS emulator), repeat the same process and create “device B personal document”.
7. At this point, device A sees “device A personal document”, and device B sees “device B personal document”.
8. Open the document on device A and add the heading: “Device A personal heading”.
9. Do the same on device B, adding: “Device B personal heading”.
10. Close the editor on both devices by clicking the checkmark in a circle.
11. Reopen the personal documents on both devices to verify that the content matches the previously entered. Close the editors again.
12. Sign up on device A using a new account. After entering the email and password, an invitation code is sent to the email. Enter the code in the app on the email verification page. After verification, the app does not yet navigate to the login page – this is expected behavior.
13. Repeat the sign-up process on device B using a different new account.

A.2.2 Team Management and Document Editing

1. Navigate to the home page and open the drawer. Verify there is no "Login" button in the drawer, confirming that the user is logged in. Do this on both devices.

2. Open previously created personal documents on both devices and verify that the content remains unchanged. This demonstrates that users can start using the app without logging in and later transition to a synchronized state.

3. On device A, open the drawer, go to "Teams" at the bottom, and click "Add team". In the dialog, enter "Demo team" and confirm. After a short synchronization, the Demo team appears above the "Add team" button.

4. Open the Demo team on device A. It shows no documents yet, and one member – the owner (the account signed up on device A). There are no invitations.

5. On device A, click "Invite member", choose "One-Time invitation", and click "Save". Copy the generated code and transfer it to device B.

6. On device B, open the drawer, go to the Teams page, and click "Join Team" in the top-right corner. Enter the code in the bottom sheet and submit. A toast confirms that the team was joined successfully. The "Demo team" now appears on device B.

7. Open the Demo team on device B. It shows no documents, two members, and no active invitations.

8. On device A, create another one-time invite. This should now appear in the invite list on both devices.

9. On device B, create a new document in the Demo team called "First team doc". Device B displays the document, marked as last edited by the device B account, with one edit recorded – this corresponds to initialization required by `appflowy_editor_sync_plugin`.

10. On device A, the "First team doc" appears. Due to a UI update issue, you may need to close and reopen the Demo team for the document details to be displayed correctly.

11. Open "First team doc" on both devices via the drawer. Initially, both documents appear empty.

12. On device A, add a heading: "Team doc heading".

13. On device B, after synchronization, add three bullet points below the heading: "Point 1", "Point 2", "Point 3".

14. After a short synchronization delay, both devices show the same content with heading and bullet points.

15. Close the editor on both devices using the checkmark and reopen the document. Check if the content was persisted.

16. On device A, create a new personal document: "device A personal document 2". Repeat the process on device B.

17. Each device now sees two personal documents tied to its respective user.
18. On device B, delete “First team doc” using the three-dot menu in the drawer. After synchronization, both devices show no team documents.
19. On device B, add a new team document by clicking “Add team document” in the drawer. Name it “Second team doc”, choose Demo team, and submit. After synchronization, both devices show “Second team doc” in the drawer.
20. Open “Second team doc” on device A and insert a heading: “Second team doc heading”.
21. Open the document on device B. Both devices display identical content with the heading.
22. Close the editors using the checkmark on both devices.
23. Navigate to the Demo team page via the drawer. Both devices display one document, the same last editor, and the same number of edits. These edits reflect batched editor changes saved to the database, as explained at the end of Section 6.1.3. The team page also shows two members and one remaining invite.

A.3 Smartwatch Application Scenario

Testing scenario of watch functionality and its synchronization with a mobile device using an Android phone and a connected WearOS smartwatch. It consists of:

- **Initialization of User Data on the Mobile Device** – Creation of data on mobile device that will be then used to test the smartwatch application and its synchronization with an Android phone.
- **Smartwatch Functionality and Synchronization** – Testing the functionality of the smartwatch application to manage tasks, track, view schedule, and complete habits. It also tests its synchronization with a mobile device.

Initialization of User Data on the Mobile Device

1. TimeNoder2 is installed on a physical Android mobile device and a physical WearOS smartwatch. All data for each app is cleared.
2. On the mobile device, walk through the introduction and create three tasks: One called “Read Harry Potter 1 book” for the project “Reading” with a deadline in a month. Another task: “Write a thesis” for the project “School” with a deadline set to yesterday. And finally, “Morning meditation” for the project “Meditation” with a deadline today at 8 AM.
3. On the mobile, navigate to the schedule page and, after the “Morning meditation” task event, add another event. Create a new event in the 9:00–14:00 range with the “School” project assigned and set it to repeat daily.
4. On the mobile, navigate to the habits page by opening the drawer and selecting “Habits.” At the top of the page, click on “Select Projects” and mark “School,” “Meditation,” “Reading,” and “Training” as habits, then confirm.

5. Click the check button and mark “Training” as completed for today.

Smartwatch Functionality and Synchronization

1. Keep the mobile app open and navigate back to the tasks page.
2. Turn on Bluetooth on the mobile and ensure that the smartwatch is connected.
3. Open the TimeNoder2 app on the smartwatch and wait for it to load. Keep the app open and wait for about 15 seconds. You should see the “Morning meditation” task listed under today’s tasks.
4. Tap “Today” at the top and select “Previous” on the selection page. You should see the task “Write a thesis,” which had a deadline yesterday.
5. Now navigate to future tasks in the same way. You should see “Read Harry Potter 1.” task instead.
6. Tap the play button on the right side of the “Read Harry Potter 1” task tile.
7. After a moment, you should be navigated to the timer page, and the timer should be running, tracking the “Read Harry Potter 1” task. The current status should be “Work – Round 1/4.” Wait until the timer runs for more than a minute, then tap the skip button at the bottom. The timer should switch to the “Break – Round 1/4” state, showing a 5-minute break. The task is still being tracked.
8. On the mobile device, check the current timer status at the top of the tasks page. It should also show 5 minutes. This verifies that the timer state was successfully synced.
9. On the mobile device, go to the schedule page. You should see a tracking record for the “Reading” project and the “Read Harry Potter 1 book” task. This confirms that the timer on the watch created the event and that it synced successfully to the mobile.
10. On the watch, tap the hamburger icon and navigate to the schedule page. You should see a similar representation of the schedule page.
11. On the watch, go back to the timer page and start the timer again. Now close the app by swiping to the right. Go to the watch home screen. You should see an activity indicator for the app at the bottom. Opening the notification panel should show the current timer status. This verifies that the timer runs independently of the app, even when it is killed by the OS – unlike apps like TickTick or Focus To-Do.
12. On the watch, open recent apps and verify that the current timer status is visible there too.
13. In the watch’s notification panel, tap the timer notification and press the skip button. Then return to the home screen and verify that the ongoing activity disappeared. In recent apps, confirm that the timer status is no longer shown in the app tile.
14. Open the timer notification again and start the timer. Verify that the ongoing activity appears again and that recent apps show the correct status. This confirms that the smartwatch can control the timer from the notification, allowing long tracking without keeping the app open.

15. On the phone, check the timer status. It will still show the old state, because the smartwatch app was not opened to allow synchronization.

16. On the watch, open the TimeNoder2 app and go to the timer page. It should show the same state as in the notification. This confirms that tracking transitions smoothly into the app UI.

17. On the phone, it should show the timer in the same state as on the watch, but in the "start" state and not running. This is intentional, as the running state only lives in one device.

18. Wait until the timer runs for more than a minute and then tap the close button on the currently tracked task tile. The timer keeps running, but no task is selected.

19. On the phone, verify that a new tracking event is created on the schedule page for the "Reading" project and "Read Harry Potter 1 book" task.

20. On the watch, dismiss the app (do not close it entirely) and verify that the ongoing activity and timer notification remain correct. The notification should not show a tracked task.

21. Close both the mobile and watch apps.

22. Reopen both apps.

23. On the mobile app, go to the timer page and press "Stop" to reset the timer to the default state.

24. On the watch, open the timer page and verify that the state matches.

25. On the watch's timer page, tap on "Project" and select the "Meditation" project, and wait.

26. After a moment, verify that "Meditation" is selected on both devices.

27. On the watch, go to the habits page and check the status of the "Meditation" habit. It should not be completed for today.

28. Return to the timer page and track "Meditation" for over one minute. Then stop the tracking by tapping the close icon on the project tile.

29. Go to the habits page and confirm that "Meditation" is marked as completed.

30. On both the watch and phone, go to the schedule page. There should be 5 events in total, with the latest one assigned to the "Meditation" project.

31. On the mobile app, go to the habits page via the drawer and verify that the "School" habit is not yet completed.

32. On the watch, scroll to the event with the "School" project and mark it as done.

33. On the phone, verify that the "School" habit is now marked as completed. Confirm the same thing on the watch's habits page.

34. On both devices, navigate to the tasks page.

35. On the watch, go to previous tasks by tapping “Today” and selecting “Previous.”

36. Mark “Write a thesis” as completed on the watch by tapping the circle on the task tile.

37. On the phone, check that it is also marked as completed under today’s completed section.

38. On the watch, go back to today’s tasks and start tracking “Morning Meditation.” Then navigate to the settings page and enable strong timer vibrations.

39. Let the timer run, close the app, and verify that the timer vibrates at the end and switches to a paused state showing “05:00.” The ongoing activity icon should disappear.

40. On the phone, go to the timer page. Tap the three-dot menu at the top right, select “Tracking mode,” change it to “Until stopped,” and confirm.

41. On the watch, stop the “Morning Meditation” tracking and start tracking the “Training” project. Verify that the ongoing activity icon appears on the home screen. Let it run, close the app, and after around two hours, check that the timer is still running and the activity is visible. Then open the timer notification and tap “End.”

42. Ensure the phone is connected to the watch. On the phone, go to the schedule page and verify that tracking for the “Training” project has been recorded.