Development of an Open-Source and Cross-Platform Client for a Collaborative List Application

Lucas Jung

Jamila Sam, Barbara Jobstmann

EPFL BA5 2023

Contents

1 Introduction **1**
- 1.1 Understanding the Problem Space . . . 1
  - 1.1.1 Survey of Existing Solutions . . . 2
- 1.2 Goals . . . 2
- 1.3 Walkthrough . . . 3

2 Tooling **4**
- 2.1 Programming Environment . . . 4
  - 2.1.1 Language and Framework . . . 4
  - 2.1.2 Used Packages . . . 5
- 2.2 Design . . . 6

3 Crafting the User Experience **7**
- 3.1 Defining the Design Ideology . . . 7
- 3.2 Application Logo . . . 8
- 3.3 User Interface . . . 9
  - 3.3.1 Account Forms . . . 9
  - 3.3.2 Collection Pages . . . 10
  - 3.3.3 Modals . . . 11
  - 3.3.4 Selecting the Right Font . . . 11

4 Code Architecture **13**
- 4.1 Project Structure . . . 13
- 4.2 Models Structure . . . 16
  - 4.2.1 Account vs. Account Properties . . . 17
  - 4.2.2 Dynamic links and reactivity . . . 18

5 Results **20**
- 5.1 Overview of the Application . . . 20
- 5.2 Beta Testing . . . 24
  - 5.2.1 Reviews . . . 24
- 5.3 Future of Open-Items . . . 26
- 5.4 Conclusion . . . 28

A Design (Figma export) **30**

Chapter 1

Introduction

In a world filled with to-do lists, shopping lists, project task lists, and more, the need for an efficient and user-friendly list making tool is definitely present. Many of us have experienced the frustration of using various list making applications that did not quite meet our expectations. This project aims to address these issues by creating a versatile and user centered list management solution called Open-Items.

1.1 Understanding the Problem Space

The motivation for this project arises from my personal experience of using a lot of different list writing tools. It ranges from traditional pen and paper to very sophisticated note taking software with fancy graph supports. However, these tools consistently fall short of my expectations. Their main problems and limitations are exposed below.

First, most of the existing solutions involve proprietary software, which consequently come with a long list of associated flaws:

- Cost, freemium (or straight up paid) plan, sometimes even ads.
- Security and transparency concerns.
- Vendor lock-in making it way harder for users to switch to a different solution if/when they want to.
- Lack of community/users consideration, input and participation.
- Single client for interacting with the application.
- Enforced application updates that break user habits.

Secondly, some solutions suffer from an excess of features, significantly reducing usability as a result. This often takes the form of the unnecessary inclusion of calendar and contact functionalities.

Finally, they often support only a single platform, forcing users to seek alternative solutions for each of their devices, with the syncing process and separation of concerns that it implies. As a result I am using a different device, often my phone, rather than the one I was working on, often my computer, to access my lists which is far from ideal.

To address these issues, this bachelor project will explore the development of a better list management application empowering users with an efficient solution for storing lists.

1.1.1 Survey of Existing Solutions

In this section we will briefly discuss some of the existing solutions in the list management space (in no particular order) to provide context for the problems mentioned above.

- Google Keep[1] (Proprietary) - It has a nice user-interface and it is easy to use. However, it does not offer offline functionality.
- Microsoft ToDo[2] (Proprietary) - It also has a good user-interface. It does not work offline and it does not support infinite item nesting.
- Todoist[3] (Proprietary) - An excessive number of features makes it very hard to use. There is still no offline support.
- Any.do[4] (Proprietary) - The user interface could be a lot simpler, and it also suffers from too many features.
- Tasks.org[5] (Open-Source) - It is the existing solution that is the closest to this project (mainly due to being open-source). It is getting old: it was based on Astrid[6] which was created in 2008 and discontinued in 2013. It is not cross-platform, only packaged as an Android application (no WEB, desktop or Apple support).
  The application does not feel intuitive as the interface is not really user-friendly. The buttons feel out of place and there is too much empty space around items. I also find the menus hard to navigate with too many options packed at the same place.
  The syncing backend choices are not ideal as different standards have limited compatibility with the application's features.

The landscape is largely dominated by proprietary applications carrying all the drawbacks listed above and lacking the option to self-host the backend infrastructure if desired.

1.2 Goals

The goals for this bachelor project are centered around describing and developing a user-friendly and efficient list management application that aims to address the identified problems in the current landscape. The primary focus will therefore lie on the development of the client-side component of the Open-Items project. Nevertheless the upcoming server side addition will be kept in mind in order to have a good basis to better integrate the backend part when ready.

[1]Google Keep: https://keep.google.com
[2]Microsoft ToDo: https://to-do.microsoft.com
[3]Todoist: https://todoist.com
[4]Any.do: https://www.any.do
[5]Tasks.org: https://tasks.org
[6]Astrid: https://en.wikipedia.org/wiki/Astrid_(application)

The development will have the following core directives:

- Simplicity: the project will prioritize simplicity in design and functionality, complexity will only be introduced as last resort when it really benefits user experience.
- Offline support: the client will adopt an "offline first" approach, ensuring that the application works as expected even when internet connection is lost. When connectivity is restored, the client will catch up with the server and synchronize any change made during the offline period.
  On top of that, the client will support fully offline "accounts", ensuring that none of their list data will ever be sent to a server. This feature provides enhanced privacy and security as the data from those accounts will not leave the device they were created on. Sharing and synchronization will therefore be disabled for those accounts.
- Open-source support: the application will embrace open-source principles and practices.
- Cross-platform operability: the application will be designed to work seamlessly across various devices and operating systems. It should be accessible on desktop computers, smartphones, and tablets, ensuring users can access their lists from any device.

1.3 Walkthrough

After this introduction, the document is split into different parts.

We will begin by looking at the tools used for the project. This part explains the programming environment, programming languages, UI framework, and the main packages used.

The third section, "Crafting the User Experience", is about the designing phase of the project. It talks about the design ideology, the application logo creation, and explains the user interface, including authentication forms, collection pages, modals, and font selection.

Next, we get into the details of the code structure. This part focuses on the project's code organization and models' structure. A detailed diagram of the interactions between the code's data classes will provide a great first understanding of how the different parts of the code interact with each other. It later goes into more details about the interesting difference between account and account properties, and the handling of dynamic links and reactivity. Please note that this section assumes prior knowledge about reactive user interfaces implementation and dynamic database references.

Finally, we get an overview of the application, and talk about the beta testing phase, before discussing the experiences and lessons learned during development, and conclude with a summary. The document wraps up with an appendix, which contains extra design resources along with the full Figma export that might interest the most enthusiastic readers.

Chapter 2

Tooling

In this chapter, I will provide an overview of the primary tools and resources that I used for this project.

2.1 Programming Environment

Choosing the right programming environment is important for any software project. I have selected a combination of tools that align with the requirements explained in Chapter 1 Introduction. I also had to consider the limited timeframe of the project and the need for simplicity, given that I was developing the application alone.

The code is version controlled in the Open-Items Git repository[1] on GitHub. I chose GitHub as it is already the most widely used online Git version control platform for open-source projects, and it had all the functionalities I needed.

2.1.1 Language and Framework

I chose to code using the Dart[2] programming language along with its Flutter[3] UI framework. Dart is a versatile and performant high-level language. Originally developed at Google in 2011, it is known for its efficiency and compatibility with various platforms. It comes with the Flutter UI development kit also created at Google in 2017. Together they make the perfect pair for developing cross-platform applications with a single code base.

Additionally, Dart comes with a modern and expressive syntax that is both easy to read and write. It supports advanced language functionalities like generics, null-safety, mixins, code generation and type inference. It comes with a comprehensive standard library and a package manager called Pub[4] providing a rich ecosystem of tools and libraries to facilitate the development even more (see Subsection 2.1.2 Used Packages).

Flutter gained popularity for its ability to create stunning and natively compiled applications for mobile, web, and desktop. It has a widget-oriented approach to building

[1]Open-Items Git repository: `https://github.com/gruvw/open_items`
[2]Dart: `https://dart.dev`
[3]Flutter: `https://flutter.dev`
[4]Pub: `https://pub.dev`

user interfaces: a widget is conceptually similar to an object for describing the state, logic, interaction and design of a visual application component. This is especially great to improve code reusability and to apply the DRY[5] (Don't Repeat Yourself) principle to user-interface design.

2.1.2 Used Packages

During the development process I used multiple essential Flutter/Dart packages in order to improve my workflow and boost my productivity. Note that all packages required to build the project are already specified in the `pubspec.yaml` file, meaning you do not have to install them manually. When you build the project, Flutter will automatically download the correct version of every package and bundle them together to form the final binary executable.

Here is a quick overview of the main packages I have used:

- Hive DB[6]: Lightweight and very fast key-value database written in pure Dart. This NoSQL database has the main advantage of being fully cross-platform without requiring any other dependency. Moreover, it integrates seamlessly with the other packages below. This package is used in the Open-Items client to persist user data across application restarts.
- Shared preferences[7]: A platform-specific persistent storage wraper for simple data. It is very useful for non-critical, interface related data persistence.
- Riverpod[8]: A reactive caching and data-binding solution, simplifying application state accessibility. It is mainly used to reactively trigger widget rerenders when the application data changes.
- Flutter Hooks[9]: Hooks are a set of reusable and customizable widgets and utilities. They simplify common tasks, increase code sharing between widgets and facilitates the process of writing stateful widgets.
- NanoID[10]: A tiny, secure and URL-friendly unique string ID generator. It is very useful for generating unique identifiers within the application. I mostly used it to reference objects in the local Hive database. This package was originally made for JavaScript where it gained a lot of popularity. It was therefore later ported to many more programming languages, including Dart.
- Icon Font Generator[11]: An easy way to convert SVG icons to an OpenType font and generate a Flutter-compatible class that contains identifiers for the icons.

This list is non-exhaustive, and it is worth noting that there are numerous smaller packages that I used throughout the project, though their significance and contribution may be relatively minor compared to the ones described above.

[5]DRY: `https://en.wikipedia.org/wiki/Don't_repeat_yourself`
[6]Hive DB: `https://hivedb.dev`
[7]Shared preferences: `https://pub.dev/packages/shared_preferences`
[8]Riverpod: `https://riverpod.dev`
[9]Flutter Hooks: `https://github.com/rrousselGit/flutter_hooks`
[10]NanoID: `https://github.com/ai/nanoid`
[11]Icon Font Generator: `https://github.com/ScerIO/icon_font_generator`

2.2 Design

For the design phase of the project, I relied on Figma[12], a powerful cross-platform design and prototyping tool. This application is really well crafted and allows for fast paced design iterations thanks to the following features:

- Vector editing - Figma's vector editing capabilities are well built. They simplify the creation of scalable and resolution independent assets that can adapt to various screen sizes.
- Components and styles - The components/styles/variables system facilitates the creation and management of design elements by maintaining coherence across every component. If you need to change a shared element, like the font size or the background color, the change will be automatically reflected across all components and pages.
- Exports - Figma provides tools for exporting assets making it easier to include designs in different places, like this document: see Appendix A Design (Figma export).
- Live preview - The live preview feature allowed me to generate a link that led to a live updating read-only view of my designs. It allowed for fast paced iterations when gathering feedback from friends and family. It also reduced the friction associated with sending updates to the two project supervisors every time I made changes. We always had the same shared and up-to-date view of the multiple design elements and application pages.

[12]Figma: `https://www.figma.com`

Chapter 3

Crafting the User Experience

In this chapter, we will dive into the design phase of the Open-Items client user interface. Designing a good user interface is a crucial aspect of any software development project, as it directly impacts how users interact with and experience the application. Over the course of about two weeks, I dedicated my efforts to drawing the different pages and UI components of the application. These designs serve as the blueprint for the user interface, and will be integrated as closely as possible into code at a later stage.

In this chapter I will frequently refer to design elements of Appendix A Design (Figma export), denoting them with unique design numbers (e.g. `D1.1` for the first image). Additionally, I will include certain images within the text for improved accessibility and convenience when navigating between them.

Keep in mind that these designs are purely creative work and may evolve when starting the development journey and taking into account the actual Flutter code implementation.

3.1 Defining the Design Ideology

The design of the application places a strong emphasis on simplicity and minimalism. The primary goal is to enable every user to accomplish tasks quickly and efficiently. Unnecessary distractions must be avoided to keep the user focused on their intended goal when opening the application. The perfect design is the one the user does not notice, as it seamlessly integrates into their workflow, making tasks feel effortless and intuitive. Paradoxically, this pursuit of simplicity can prove to be surprisingly complicated in practice.

To achieve this, the design consists of a bichromatic color scheme of black and white. This choice ensures the highest contrast making it easier to read and navigate, while mimicking the list writing experience that everyone is used to on pen and paper. There are only a handful of exceptions to this rule, being some shades of grey for placeholder texts like `D5.3`, and the use of red indicating a potentially dangerous or irreversible action is about to take place like in `D3.2` and `D4.2`. This distinctive "danger" color, being the only one out of the black to white spectrum, makes it an even greater contrast effectively warning the user.

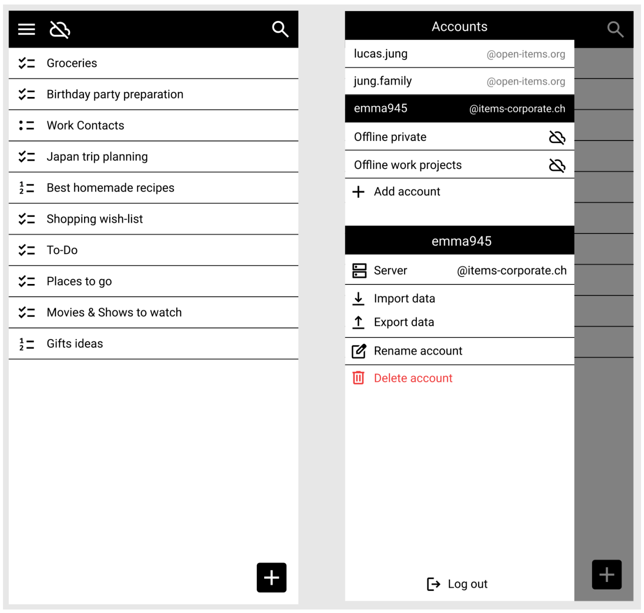


Figure 3.1: D3.1 - Lists page & D3.2 - Side bar

Most of the icons used in this design are taken from the Open-Source Google Icons[1] library. These icons align with the project's design principles and they are already integrated into the Flutter ecosystem, making this icon library a good basis for the project. In the rare instances where I could not find the icons I needed in this library, I designed my own in the same "Material" style directly in Figma.

3.2 Application Logo

The application logo also had to adhere and comply to the principles of simplicity and to the bichromatic color scheme in order to effectively reflect the overall design and feel of the application. Furthermore, given that an application logo occupies only about 1 percent of the total available home screen space on a standard smartphone, it had the additional constraint of being visible and recognizable even at a reduced size.

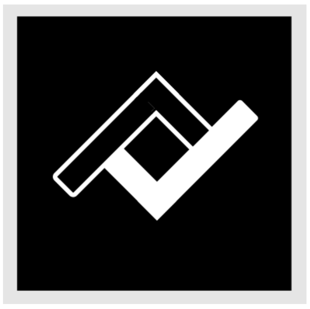


Figure 3.2: D1.1 - Launcher icon dark

The concept of the logo involves interesting ideas hidden behind a minimalistic design. It features a basic checkmark symbolizing the completion of a checklist item, stacked with

[1]Google Icons: https://fonts.google.com/icons

the reversed outline of a second checkmark on top of the first one. This combination also forms a tilted empty checkbox at the center for those who pay close attention to details.

The `D1.1` design element (on page 31) illustrate the step by step process of how the logo was put together from left to right. It starts with the single checkmark drawing, followed by the beforementioned combination of the two checkmarks, and finally, on the very right, the two variations of the logo (with reversed colors). After gathering feedback from my friends and family, the dark background version was the most popular one, so I decided to stick with it.

3.3 User Interface

The design of the user interface is divided into three main categories, each of which is discussed below.

3.3.1 Account Forms

When users open a complex application, their initial step is typically to create or log into an account. There are three types of forms that serve this purpose before one can fully use the application.

1. Create an online account `D2.1`
2. Create an offline account `D2.3`: the only account type supported for the bachelor project, as it will only support client-side operations.
3. Log into an existing online account `D2.2`

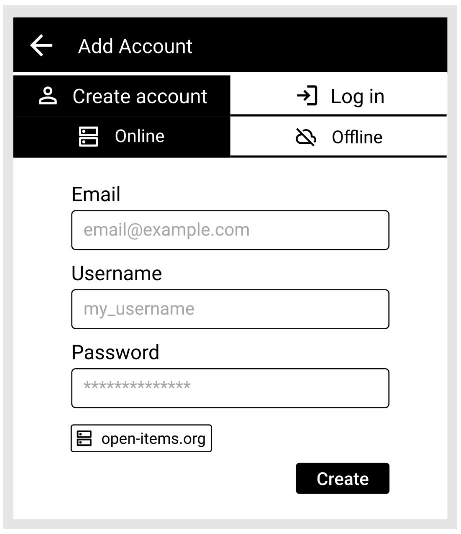


Figure 3.3: D2.1 - New online account form

The two online account related forms offer the option for users to select a hosting server for their account. This is possible thanks to a server selection component where one can manually set the URL/IP to use, permitting self-hosting capabilities. This custom server notion is also present in `D3.2` where connected accounts are listed, as a single client could have access to multiple accounts stored on different servers.

Additionally, as the usual account creation forms do, the user's email address is required making mass account creation more difficult and enabling account password recovery.

3.3.2 Collection Pages

This is probably the most important part of the user interface as it is where the users will primarily interact with the application.

The lists page `D3.1` is the first screen that will show up when the application is opened, provided there is at least one active account. It displays the different lists the user has along with icons depicting the list type (checklist, ordered list, bullet points).

Once a user clicks on a list the application will transition to the items page `D4.1`, showing the different items in the selected list. As already mentioned, one of the unique features of the application is "infinite nesting": every item can be a collection in and of itself and have subitems attached to it. When an item has subitems, the item's text is displayed in bold, and clicking on it leads to `D4.2`, which shows the subitems.

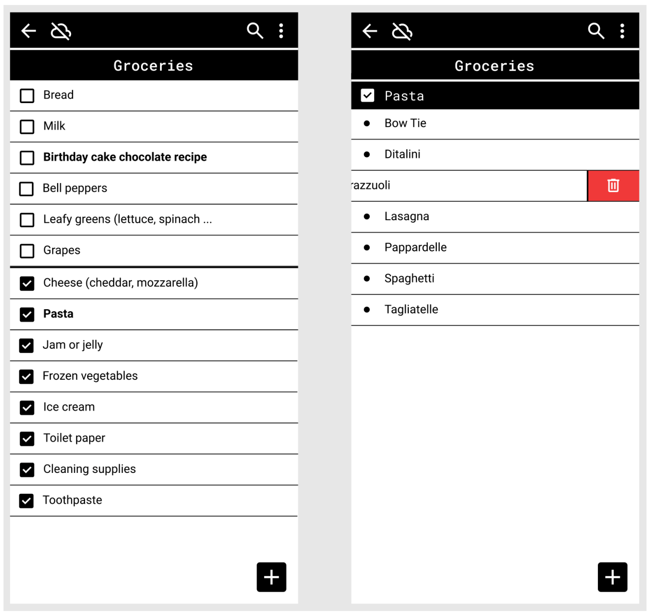


Figure 3.4: D4.1 - Items page & D4.2 - Subitems page

In all of these pages, there is an new/add button at the bottom of the screen to add new elements to the current view. Users can also slide elements from right to left to reveal the delete button (e.g. `D4.2`), which you then need to press to actually delete the collection: a non-invasive double action used for deletion confirmation.

Reordering collections is achieved through a long press and dragging them to the desired position. Collection specific settings and options are accessible through the "more" menu, represented by three vertically stacked dots in the top right corner of the screen. The offline icon indicates that the collection currently displayed is local to the device and not synced to any server (offline account feature).

3.3.3 Modals

Finally, the pop-up modals play the crucial role of informing and interacting with the user in a synchronous manner.

- `D5.1` - A simple information dialog, making sure the user has read, agreed to and/or understood a specific message.
- `D5.2` - A cancelable information dialog asking for extra user input or providing additional context before actually executing or canceling a predefined action.
- `D5.3` - A single text input field form to acquire a string input from the user.
- `D5.4` - A selection dialog allowing the user to choose one option among several propositions.
- `D5.5` - The ordering options modal letting the user choose their preferred way of ordering their collections. This one is a bit different from the others as it appears from the bottom of the screen, letting the user see the chosen ordering apply to the collections as they select it.

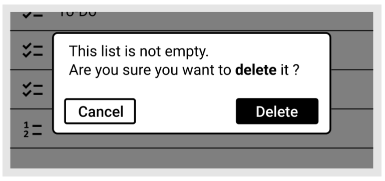

This list is not empty.
Are you sure you want to **delete** it ?

Figure 3.5: D5.2 - Cancel dialog

3.3.4 Selecting the Right Font

Choosing the appropriate font for the application is important as it impacts readability, ease of use and user engagement, making it a central point in shaping the application's visual identity.

I restricted the fonts to choose from to the Open-Source Google Fonts[2] library, as it has a lot of choice and is already integrated into the Flutter ecosystem.

I initially considered using a monospaced font, as it would perfectly align with the application's design principles of readability and ease of use, often disambiguating similar characters and not having the overhead of reading variable width letters. On top of that, as a developer I personally always use them and find them appealing. However, after gathering feedback from friends and family, it became evident that they found monospaced fonts too disruptive as they were not used to seeing them. They felt like it was giving an old/typewriter feel to the overall application which was not the initial purpose. Another factor that came into play is the fact that monospaced fonts tend to use more horizontal space than proportional fonts. This can be an issue on smaller devices.

[2]Google Fonts: `https://fonts.google.com`

The ultimate goal for the chosen font is for it to seamlessly blend into the user experience, becoming virtually unnoticed. After a lot of hesitation between enforcing a monospaced font, potentially causing slight disruption to users for improved readability, and the imperative that the font should go unnoticed, I finally decided to opt for the widely popular proportional font, Roboto[3] designed by Christian Robertson and licensed under Apache License v2[4].

[3]Roboto: `https://fonts.google.com/specimen/Roboto`
[4]Apache License v2: `https://www.apache.org/licenses/LICENSE-2.0`

Chapter 4

Code Architecture

Let us dive into the organization of the codebase, examining the directory structure and providing insights into the purpose of each major component.

4.1 Project Structure

The project follows a strict and well-defined directory structure to enhance clarity, separation of concerns, abstractions, and maintainability. Here is a nonexhaustive overview of the primary directories:

```
global
└── styles
    └── icons

models
├── objects
├── ordering
├── properties
└── stores
    └── hive

state
├── application
└── shared_preferences
```

```
utils

widgets
├── authenticate
├── collections
│   ├── collection_page
│   └── lists_page
│       └── drawer
├── components
│   ├── buttons
│   ├── modals
│   └── structure
├── router
├── utils
└── validation
```

Let us navigate through the directories and files so as to understand the purpose and interconnections between every element:

- The `global` directory serves as a container for globally used resources and configurations. This includes global `styles`, colors, `icons` and layout constants that are utilized throughout the entire application. Keeping these resources centralized ensures consistency in the user interface, avoids duplication and simplifies potential future modifications.
  This directory also contains files related to all the constant string literals and core constant values that might be shared across the codebase.

- The `models` directory is dedicated to defining the data models used within the application. This includes the abstract structure of objects, ordering mechanisms, and various properties associated with the data. The models in the project are described in greater details in the Section 4.2 Models Structure below. It also contains the database abstraction class and implementations that are used throughout the whole codebase.

  - The `stores` subdirectory holds the concrete implementations of the models for device local persistence, specifically using the `hive` database.

  - The `ordering` subdirectory deals with mechanisms related to sorting and ordering data. It involves sorting algorithms, comparators, or any logic associated with the arrangement of collections. It is also in this directory that the two enumerated types related to ordering (`ListsOrdering`, `ItemsOrdering`) and their variants are defined.

- The `state` directory contains the implementations of the application's live state management. This ranges from global `application` state, authenticated accounts, collections, and any other user interface state related logic. It contains the definitions of all the providers used for application reactivity (see Subsection 4.2.2 Dynamic links and reactivity) and the different database middlewares allowing UI rebuilds when subscribed data changes.

  - The `shared_preferences` subdirectory contains code related to managing shared preferences, which are used for persisting simple key-value pairs such as non-synced user selected application settings (like the default collection type).

- The `utils` directory contains general purpose functions and helper classes that are not complex enough to justify a dedicated module. It contains for example some Dart language extensions for convenience: specific workarounds over some dart-lang[1] related open issues.

- The `widgets` directory houses every single UI component that contributes to the overall structure and appearance of the application. These components are designed to be modular, reusable and self-contained. As such, most of the files in this directory are making extensive use of all the functionalities present in the other directories described above.

[1]dart-lang: `https://github.com/dart-lang/language`

- The majority of the atomic design components from Figma are implemented in the `components` subdirectory. It is a versatile collection of components that are used across various parts of the application.
- The `utils` subdirectory serves the same purpose as the top level directory with the same name, except that it is geared towards user interfaces. It holds some hook definitions, the overall scrolling behavior, and some progress indicators for example.
- The `router` subdirectory is responsible of the navigation in the application. The route generator is implemented there, and whenever the user requests to view a new/different page it is code from this directory that gets executed. It is also responsible for displaying the correct initial page when the application is opened.
- The `validation` subdirectory is the place where all input validation is implemented. The core validation library is defined in the `core.dart` file. The other files in this subdirectory are specific validator implementations for different kind of user inputs (account names, list titles, item texts, ...).
- There are a total of three pages/screens in the application. The authentication page is defined in the `authentication` subdirectory. The list page is defined in the `lists_page` subdirectory along with its account sidebar in `drawer`. Finally the collection page, responsible for displaying items (or subitems), is defined in the `collection_page` subdirectory.
  All of these require some closely feature-related UI components that are implemented in their respective directory. It facilitates a nuanced comprehension of the code at a local level, mitigating the need for extensive back and forth that would otherwise be required when working on a specific page.

4.2 Models Structure

The following diagram accurately represents the Dart code abstractions and aims to efficiently cover the diverse data models within the project.

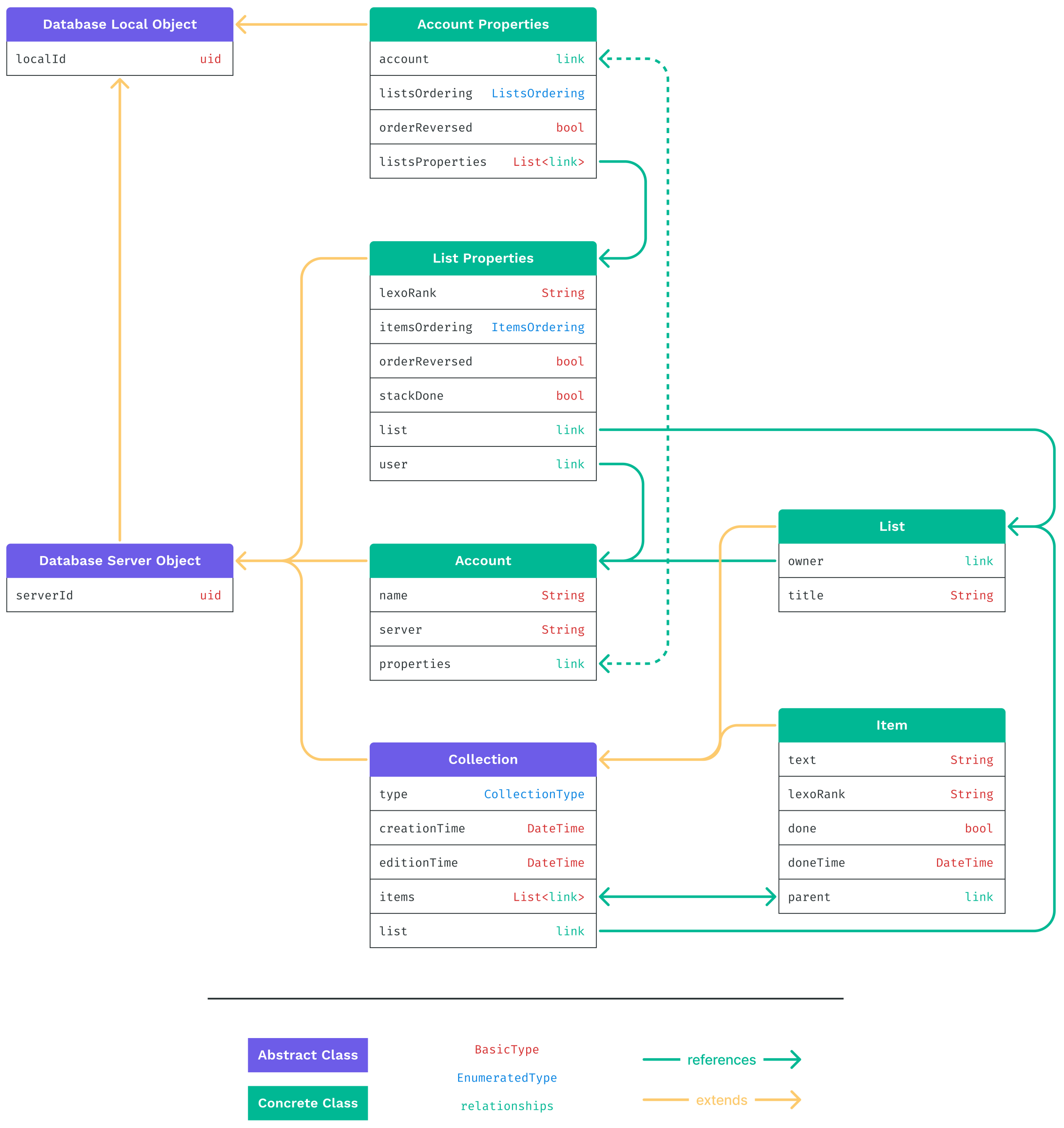


Figure 4.1: Code models diagram

Here is a brief overview of the primary classes depicted in the diagram:

- `Database Local Object` - An abstract class representing any object that needs to be persisted on a user's local device.
- `Database Server Object` - An abstract class that is extended by every resource that also lives on the server's database.

- `Collection` - The super-class of `List` and `Item`, compiling common functionalities and constituting the capacity of an object to reference a collection of items. Any `Item` may also be a collection of items itself. This feature is called "sublists".
- `List` - It is the entry point to a collection of collections. We can see a `List` as the most top-level item class: used to represent an item that does not have a parent collection.
- `Item` - A class to represent an element of a list/collection.
- `Account` - Represents the unit of authentification: an account is a user from the point of view of the server. Of course, in reality a user can have multiple accounts, but the sever will never be aware that two accounts belong to the same real-world user.
- `Account Properties` - The properties/metadata attached to an `Account`. It serves as a connection between an account and the lists properties it has access to.
- `List Properties` - The properties/metadata attached to a `List`. It serves as a connection between some list properties and the actual list behind it. They are required as two different accounts could have access to the same underlying list but might require their own personal attached metadata.

Let us dive into some specific intricacies surrounding key points and explore some of the more complex components in greater detail.

4.2.1 Account vs. Account Properties

You might question why the Account Properties class does not extend the Database Server Object class, unlike every other concrete class. This is due to the fact that, in contrast to the client-side, an Account and its associated Account Properties collectively constitute a unified entity within the underlying Open-Items backend server architecture.
Indeed, the `serverId` of a specific Account is the same that is used in case we need to target the corresponding Account Properties when communicating with the server. Consequently, the Account Properties cannot be strictly qualified as a Database Server Object, given that it is not associated with a different `serverId` than its Account and does not constitute a full-fledged server object on its own.

Now, you might be curious about the reason for keeping it distinct from the Account class? This has to do with the fact that not every account reference, on a particular device, belongs to the authenticated user. Therefore the user should not be able to access or store the corresponding Account Properties from this device. There are two categories of accounts that can live on a user's device: an "authenticated account"[2] represents an account that is authenticated and therefore owned by the user, whereas a "referenced account" is a plain reference to an account the local user might not have access to. Indeed, retaining a reference to an account that we do not necessarily own can be essential, for instance, to identify the creator or participants of a shared list.
This is the reason behind the dashed arrow representation of the bidirectional reference between an account and its properties on the diagram: the relationship might not exist

[2] Authenticated accounts are called `local accounts` in the codebase.

locally if the user is not authenticated with that account. This emphasizes the fundamental distinction between an authenticated account and a referenced account for which no permissions are granted.

4.2.2 Dynamic links and reactivity

In the architectural design, every concrete class in the diagram is linked with a Hive database implementation to enable object persistence (see Section 4.1 Project Structure). This standardized approach to database implementation enhances the overall robustness and reliability of the system. In addition, these concrete abstractions maintain the flexibility of swapping the database implementation with a different one without heavily refactoring the rest of the codebase.

The final element that is worth further explanations is the actual implementation of dynamic links/references depicted in the diagram. When a button is clicked on the user interface, the relevant data is retrieved from the local database. However, at this point, the view is non-reactive, merely displaying a snapshot of the data.
The challenge lies in efficiently accessing referenced objects and ensuring that copies remain synchronized, reacting to changes to consistently display the most accurate version of the data on the screen. Let us split those two concepts.

References under the hood

Each reference or link in the diagram actually corresponds to the `uid` (unique identifier) of the referenced object. It is, in some sense, comparable to a join ID column in the SQL world. However, the link system implements lazy loading: querying an object from the database yields a Dart "copy" of the object along with its properties, deferring retrieval of the referenced object.

This approach results in really fast and precise queries where we only fetch the data we need from the database. It is important as the user does not want to wait for the data to load. However, the downside is that accessing the referenced object requires a subsequent query, demanding careful model design on references to maximize performance gains. As a result, the issue transitions from being a runtime concern to being a software architecture problem.

Another consequence and challenge introduced with that kind of granularity is how to manage deletion of objects. By blindly deleting objects we do not need anymore, we might as well be leaving dangling references that point to non-existing objects everywhere in the database. It can introduce a whole new class of bugs and crashes in the application, but more importantly, it inflates the database size over time as we continually store objects that are not used and that will never get deleted.

The way I chose to address this problem is to attach an abstract `delete` method, on all Database Local Objects, that is left to be defined by every specific database object implementations of the concrete classes. The implementation of such deletion mechanism is database dependent and involves a few casting operations in non-exposed types so it has to be hidden under the Hive database implementation layer, written in the `models/stores/hive` subdirectory. Therefore, every object is also responsible of deleting all its associated referenced objects upon invocation of its `delete` method.

Both the responsibility inversion and abstraction layer allow for great separation of concerns. They keep the user interface code completely separated from this problem. Indeed, the concrete database stores keep track and manage all the references so as to make sure one does not end up with links to deleted objects.

Reactivity

We are left with the reactivity challenge: how to ensure the user interface code reacts and rebuilds widgets when the referenced displayed object properties change? We need a reactive state management solution, composed of two parts: providers and consumers.

With providers, we are solving two issues at once. First the user interface code never interacts directly with the database, as it would violate the database abstraction and separation of concerns. Instead, it always retrieves/consumes data through the corresponding provider. Additionally, this allows for providers to emit new data, effectively notifying the consumer that the data has changed, and consequently triggering widgets rebuilds. This approach greatly simplifies the user interface code with a declarative API, abstracting the complexity behind the provider-consumer interface.

A provider is able to communicate a change to its consumers by using a notification system: the database leverages a `StreamController<Event<DatabaseObject>>` called an event controller. It is a powerful data structure capable of holding a stream of objects, called events, that can be consumed at multiple places in the code. Each Database Local Object has a `notify` method that adds a new `Event` to the event controller. Every provider continuously listens to this stream of events and informs all consumers when it observes an event in the stream about the provided object.

As this piece of code is common across every provider, we can even modularize and isolate this functionality itself inside a provider: the `objectEventsProvider`. Given an object local `uid`, its responsibility is to notify the consumers when a new event about the specific object is emitted in the event controller stream. As every provider can also be a consumer, all database related providers first subscribe to one or more `objectEventsProvider`. All of this allows for a clean and highly readable syntax on the consumer side, isolating complexity once again where it is makes the most sense.

Chapter 5

Results

5.1 Overview of the Application

Welcome to Open-Items, let us take a look at the final application and how it works!

Account creation

Upon opening the Open-Items application, your first step is to create an account through the authentication form:

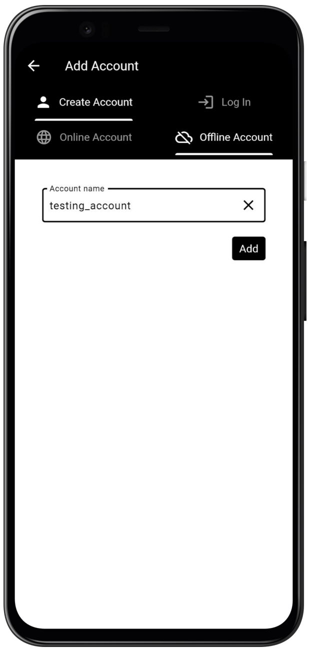


Figure 5.1: Account creation, authentication form

Home screen

Once logged in, you will land on the application's home screen, where you can interact with your lists:

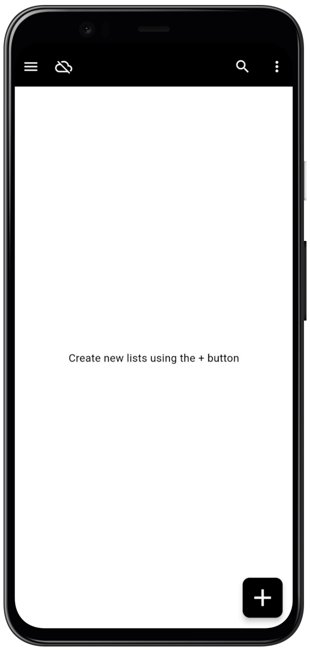


Figure 5.2: Empty account home screen

Creating a new list

Since you are starting fresh, create your first list by tapping the add ("+") button at the bottom right of the screen. Enter the title of your choice in the list creation modal:

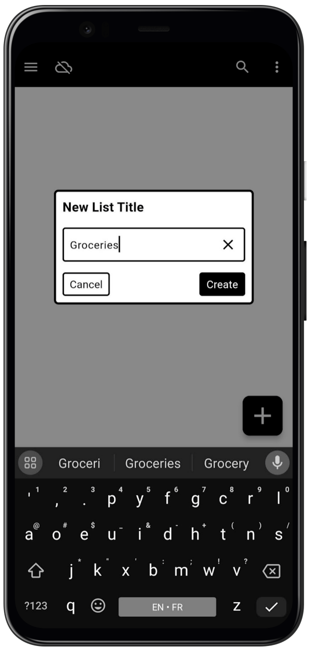


Figure 5.3: List creation modal

Your newly created list now appears on the home screen:

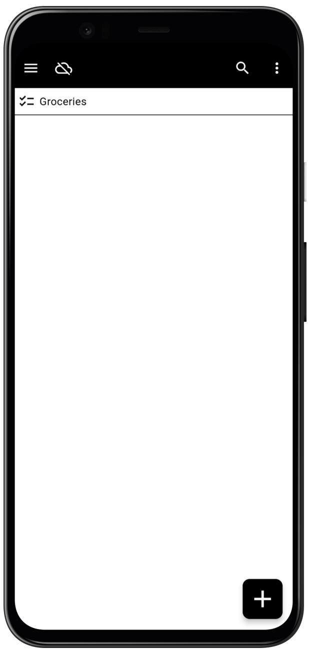
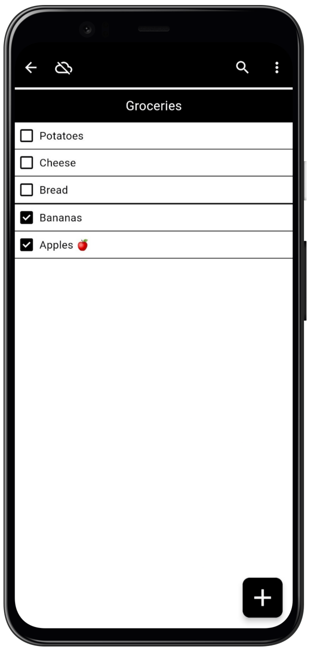


Figure 5.4: New list on home screen

You can change the default list type from the top right “More” menu.

Viewing and interacting with lists

You can now click on your newly created list to view it. To add items, simply tap the “+” button again. You can add as many items as needed. Mark items as completed by clicking on the checkbox, which automatically moves them to the end of the list.



Groceries

- ☐ Potatoes
- ☐ Cheese
- ☐ Bread
- ☑ Bananas
- ☑ Apples 🍎

Figure 5.5: List view with items

Customizing lists

Swipe from right to left on any item or list to reveal a deletion button that you can click to delete the collection. For additional customization, explore the “More” menu by clicking on the button in the top right corner. From there you can adjust item order, list type, decide whether completed items should stack at the end or not, and more:

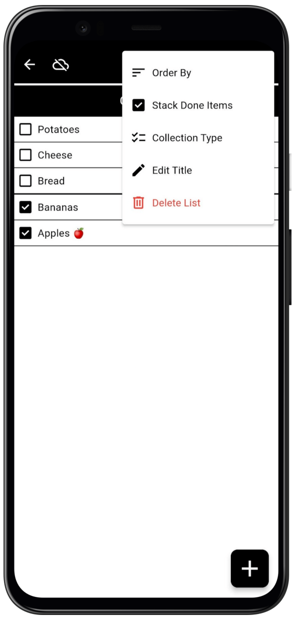


Figure 5.6: List “More” customization menu

Application sidebar

From your home screen you can also tap on the top left icon to reveal the application sidebar.

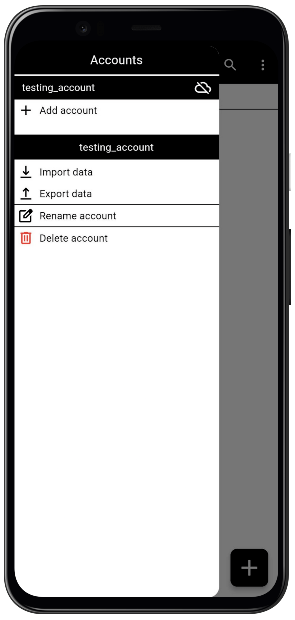


Figure 5.7: Accounts application sidebar

Here, you have control over your account settings. Do you need to manage multiple aspects of your life or do you want more control over the organization of your lists? By clicking on the "Add account" button will bring back the authentication page to create a new account. This is perfect for separating your professional and personal accounts for example.

5.2 Beta Testing

After finishing the last bit of code for the foundational aspects of the Open-Items flutter client, I could not wait for actual users to try it out and gather feedback on my work.

I set up a beta testing version of the application once all the absolutely fundamental functional aspects were working correctly. I provided two ways of trying the application:

- I hosted a static WEB version of the application on GitHub pages. I built the flutter WEB bundle (with PWA[1] support) of the Dart code and pushed it to the `gh-pages` branch on the GitHub repository. After configuring the DNS of the Open-Items domain, and linking it with the GitHub repository, everything was fully functional.

- I also built an Android APK of the application for people to try out a "native" version and to make sure it was working on a variety of Android devices. I uploaded the APK on an alpha release of the GitHub repository to host the file and provide an easy download link for everyone.

The WEB version definitely was the preferred choice, likely due to its versatility across various devices, ranging from phones to desktop computers, essentially anything that can launch a WEB browser. However I also got a few people to try the native Android application, and I received consistent and positive feedback with no notable differences or issues reported between the two options.

I added the install instructions and a few remarks on the beta testing section[2] of GitHub's project README. You can actually navigate to the link above if you want more details about the beta testing or if you want to try it yourself.

5.2.1 Reviews

Here are a few reviews I have gathered from friends and family that have tried the application:

- "This software provides a well-rounded service in a straightforward and intuitive way. The layout is smart and simple but still allows for precis organisation. The one regret I have is not having more control over the aesthetics of the app, such as color choice per list for example."

- "The application works well and is easy to use. The customization of lists is good; it could be interesting to add other designs for a commercial application." (translated)

[1]PWA: `https://en.wikipedia.org/wiki/Progressive_web_app`
[2]beta testing section: `https://github.com/gruvw/open_items#beta-test`

- "Very cute app, minimalistic in terms of design, very practical for all kinds of notes, and routine to-do lists. Really like it!" (translated)
- "Fairly instinctive and simple to use, the interface is minimalist but contains all the necessary information. It is a shame that more features are not implemented; it is a bit limiting in its usage. Overall, it is usable and nice." (translated)

Overall the feedback was very positive! It is important to keep in mind that these reviews are probably highly biased as they come from people who know me well.

Addressing common remarks

- Design personalization - One of the goal of Open-Items was to keep its design as minimal as possible to avoid distraction, a concept I went through in great details in Section 3.1 Defining the Design Ideology. I understand that people are used to being able to customize the look of their interfaces as it is a common feature among most of the top note taking applications.
  Nevertheless, I perceive these cosmetic customization options as more of a potential distraction, diverting users from their primary objectives and consuming time that could be more effectively invested elsewhere. After all, you cannot really customize a traditional piece of paper either, and it has nonetheless demonstrated remarkable effectiveness, even without it.
- Missing features - A number of users have expressed frustration due to the absence of certain features in the application. This discontent was primarily caused by the "Not implemented" dialog, which appeared when attempting to access or use features that were still in progress and, consequently, not fully available.
  To mitigate this issue, I had diligently documented all the functionalities affected by this in the beta testing instructions, aiming to prevent users from misinterpreting them as bugs. Regrettably, it appears that many users did not read through the entirety of the instructions, leading to continued confusion and annoyances.
  I perceived those comments as positive indicators that the development was progressing in the right direction. It reinforced my confidence in accurately identifying the use cases that users desire in a list application. It also serves as valuable feedback that pushes me to prioritize the implementation of these awaited features (more on that in Section 5.3 Future of Open-Items).
- Text edition - Three testers have also reported that item text edition was initially unintuitive. To change the text, you first have to click on the item, navigating to the item's page, and then click on the text to edit it.
  I must admit that having to click twice to edit the item is not ideal. The challenge is that all the other natural events were already assigned: single click to go to the item's page, long click to reorder, and a swiping from right to left reveals the delete button.
  The only remaining alternative was to swipe from left to right to reveal an "edit" button. However this approach is also not very convenient, and it requires two actions: a swipe and a click. That is without mentioning that adding another swipe might cause problems, especially when one has a lot of items on their screen. Indeed, users could accidentally trigger the swiping action, when they actually intended to scroll down to reveal lower items.

5.3 Future of Open-Items

As with most of my projects, the only thing that I would wish for is to have more time. In fact none of my projects are ever really done, they are just due. This one is no exception.

Open-Items has a bright and wonderful future ahead! All the work I contributed during this bachelor's project serves as a solid fundamental basis to grow upon.

A substantial part of my work was to meticulously plan the future features for integration into the application, and consequently structure the code to better accommodate their addition. I have thoughtfully included placeholders, ready to be later filled when integrating the associated functionality.

Let us go through some features planned to be added soon.

Import and export

One important feature is the ability to import and export collections. Users will be able to export and import data in JSON format. This will provide a standard and convenient way to save and backup your collections, and later import them back across different instances of Open-Items , or any Open-Items parser compatible application.

Additionally, a "copy to clipboard" feature will be implemented, allowing users to efficiently export collections for use in other applications.

Search collections

The inclusion of a powerful search functionality is on the horizon, with a focus on fuzzy finding[3]. This feature will enhance user experience by enabling quick and flexible searches within collections, making it easier to locate specific items.

Custom reordering with lexo-ranking

To provide users with more control over their collections, a custom reordering feature will be introduced. Leveraging lexo-ranking, users will be able to define their preferred order for items within a collection, offering a personalized and intuitive organization method.

Lexo-ranking proves to be a robust method for arranging items in a list. The mechanism involves assigning a string property, referred to as the "rank", to each item in the list. To establish the correct order, a lexicographical sort is performed by comparing the ranks of the items (like in a dictionary). When repositioning item A between items B and C, the process is simplified by updating A's rank to the average of B's and C's ranks: $R_A = (R_B + R_C)/2$. Notably, the value of a rank in this context is the base-26 representation of the rank, with each number character denoted by a letter of the alphabet.

This novel technique proves highly advantageous in the context of ordering synchronized lists. It facilitates the updating of a single property when repositioning or inserting an item. In contrast, conventional index-based ranking systems necessitate updating multiple indices across numerous items, presenting significant challenges when dealing with longer lists. This challenge is exacerbated when maintaining synchronization across multiple

[3]fuzzy finding: `https://en.wikipedia.org/wiki/Approximate_string_matching`

devices. Having a single update operation allows for a drastic reduction in the number of network packets required and eliminates most of the server side overhead.

Online synchronization with a backend server

For users who value accessibility across multiple devices, an online synchronization feature with a backend server is in the pipeline. This will ensure that changes made on one device reflect seamlessly on others, creating a synchronized and up-to-date user experience. That feature will also empower users with collaboration functionalities as they will be able to share their list and manage accesses of different accounts.

Adding this will probably take a considerable amount of work and requires careful planning and reflection to guarantee a smooth and secure user experience. To achieve this, robust encryption protocols and secure authentication mechanisms will be integrated to safeguard user data and maintain privacy. Incorporating collaboration functionalities adds another layer of complexity. User permissions, access controls, and version tracking are essential components to enable smooth collaboration without compromising data integrity.

Despite the emphasis on online synchronization, the development will strongly prioritize an “offline-first” strategy. Recognizing the importance of user accessibility in limited or in the absence of internet connection, the system will prioritize local persistent storage and offline capabilities. This ensures seamless interaction, changes, and access to information even when disconnected from the backend server. When back online, all changes made offline will be synchronized.

Others

Several other exciting features are in the works, including End-to-End Encryption (E2EE) for enhanced security, the ability to archive lists, the option to restore deleted lists locally, and the ability to move and transfer collections from account to account. These additions aim to further enrich the Open-Items experience, making it an even more versatile and user-friendly tool for personal and collaborative task management.

5.4 Conclusion

Open-Items holds a special place in my developer journey, marking a meaningful step forward. The encouragement from users during beta testing and the helpful feedback I received are driving forces for the ongoing improvements of the application.

The goals outlined in the project's initial phase (see Section 1.2 Goals) have been successfully achieved in the current version of the Open-Items client. The focus on simplicity, offline support, Open-Source principles, and cross-platform operability were all implemented, and I am satisfied with the resulting application.

The upcoming features are designed to make Open-Items even more practical and user-friendly while maintaining the cross-platform availability. Going forward, the emphasis will remain centered on efficiency and minimalism for both individual and collaborative task management. It is encouraging to see how Open-Items is becoming a tool that resonates with users' needs and embraces the Open-Source philosophy.

As I continue working on enhancing Open-Items, I am eager to explore and implement innovative solutions that streamline user experiences. I anticipate that each tweak and addition to Open-Items will not only address immediate needs but also contribute to a robust and efficient platform.

Moving forward, I am excited about the learning and growth that awaits, recognizing that every challenge brings valuable lessons and each opportunity propels me further in my developer journey. The Open-Items project is not just about coding. It is a dynamic process of shaping a tool that genuinely adds value to users' lives and hopes to improve upon existing solutions.

Experience and Key Learnings

Throughout the development of Open-Items, I gained valuable experience and insights that have contributed to my growth as a software developer. Here are some of my key learnings from this project:

- Designing a user-friendly and efficient interface requires more than aesthetics. It also demands a deep understanding of the problem space, user behavior and preferences. Learning to put into practice an iterative design process on Figma proved to be essential in refining the Open-Items user interface.
- Striking a delicate balance between simplicity and functionality is an ongoing challenge. The project underscores the importance of providing powerful features without compromising the application's accessibility and ease of use.
- The inclusion of placeholders and structural considerations in Open-Items' code lays a strong foundation for later extending the project. It should allow for seamless integration of new functionalities as the application evolves.

Acknowledgements

The development of Open-Items was made possible through the support and contributions of several individuals to whom I extend my sincere appreciation:

- Supervisors - I express gratitude to Dr. Jamila Sam and Dr. Barbara Jobstmann for their guidance. The weekly meetings we had together provided invaluable insights and a constructive space for addressing challenges and finding solutions.
- Beta Testers - I am grateful to everyone who participated in the beta testing phase. Their participation, reviews, and constructive feedback played an important role in refining Open-Items and shaping its user-centric features.
- Special thanks to my partner, Sarah Jabaudon, for her continuous support throughout the development process. Her feedback on the application (even before beta testing) and assistance in writing and refining this document were very helpful.

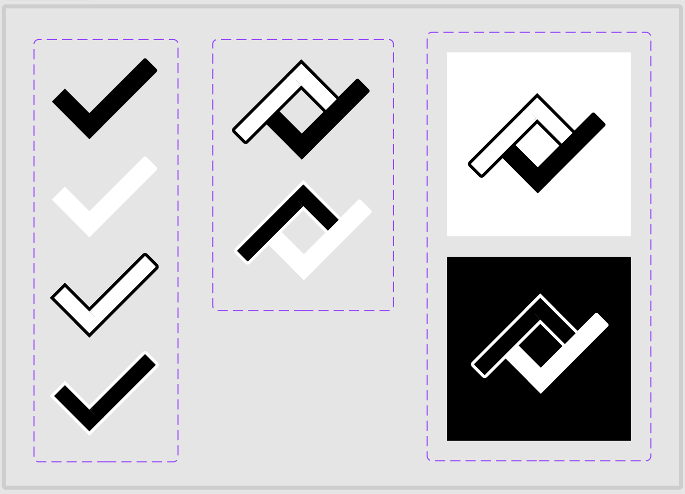
Appendix A

Design (Figma export)

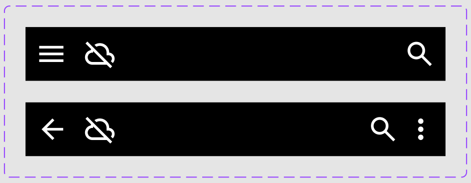
Design 1

Launcher Icon

D1.1 – Launcher icon

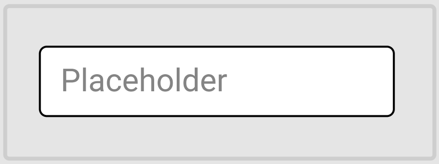


D1.2 – Application bar



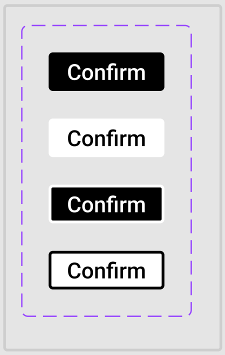
Input

D1.5 – Text input field

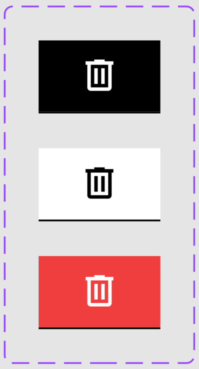


Buttons

D1.3 – Outline buttons

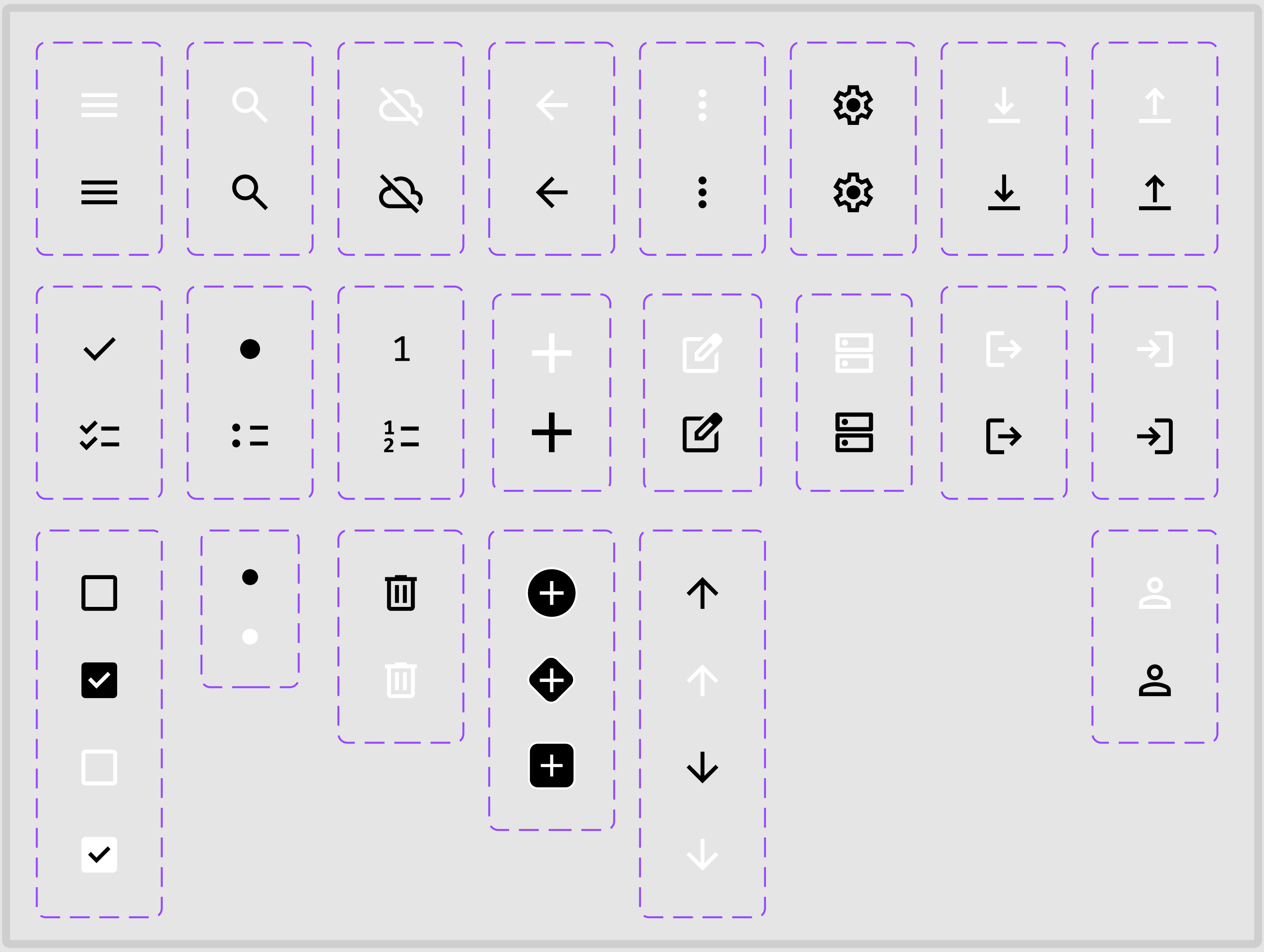


D1.4 – Delete button



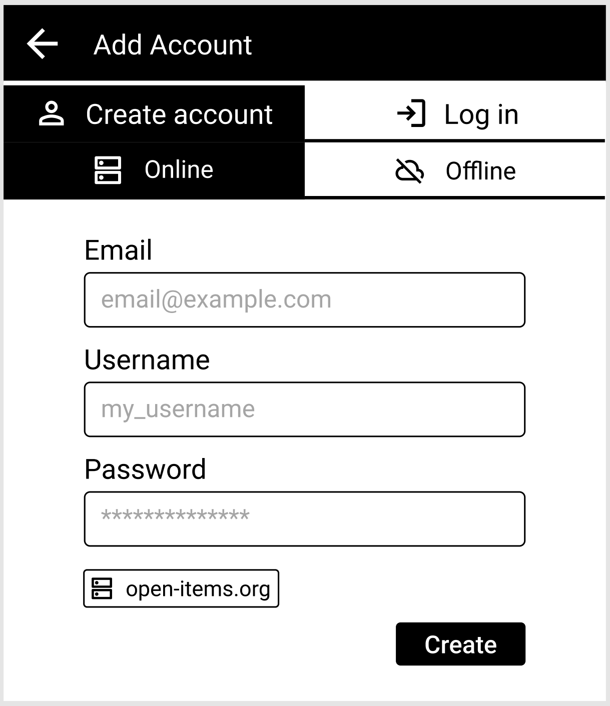
Icons

D1.6 – General icons



Design 2

D2.1 – New online account form



Add Account

Create account | Log in

Online | Offline

Email

email@example.com

Username

my_username

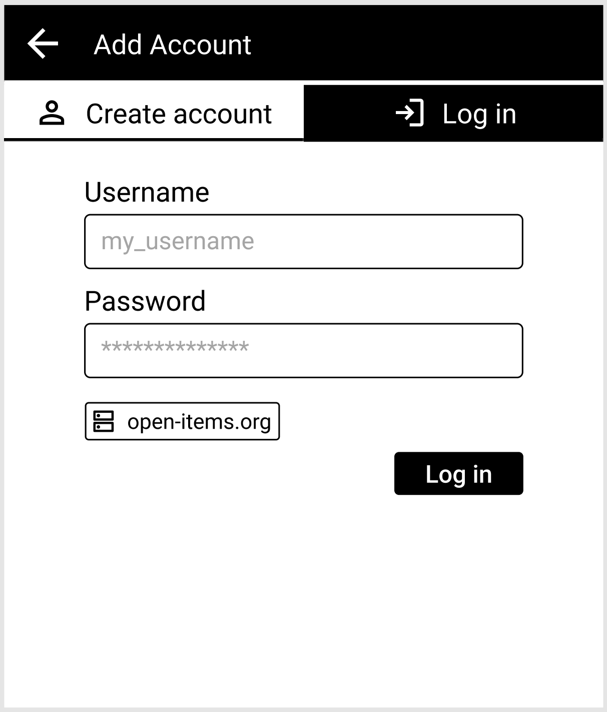
Password

****

open-items.org

Create

D2.2 – Online account login form



Add Account

Create account | Log in

Username

my_username

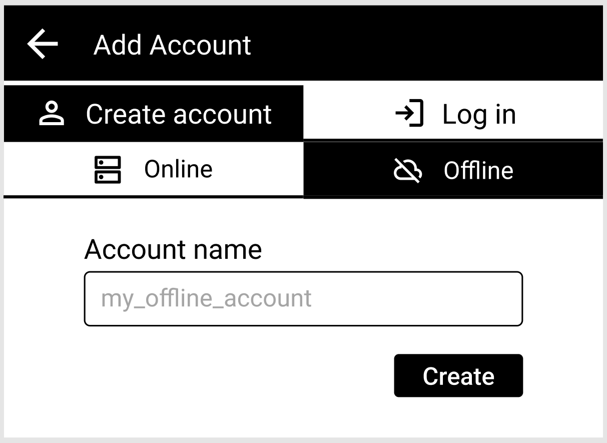
Password

****

open-items.org

Log in

D2.3 – New offline account form



Add Account

Create account | Log in

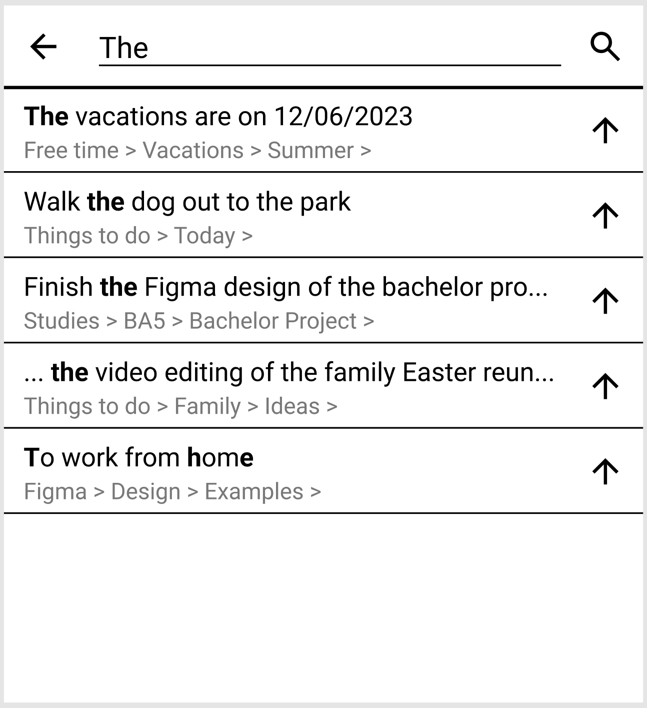
Online | Offline

Account name

my_offline_account

Create

D2.4 – Search page



The

The vacations are on 12/06/2023
Free time > Vacations > Summer >

Walk **the** dog out to the park
Things to do > Today >

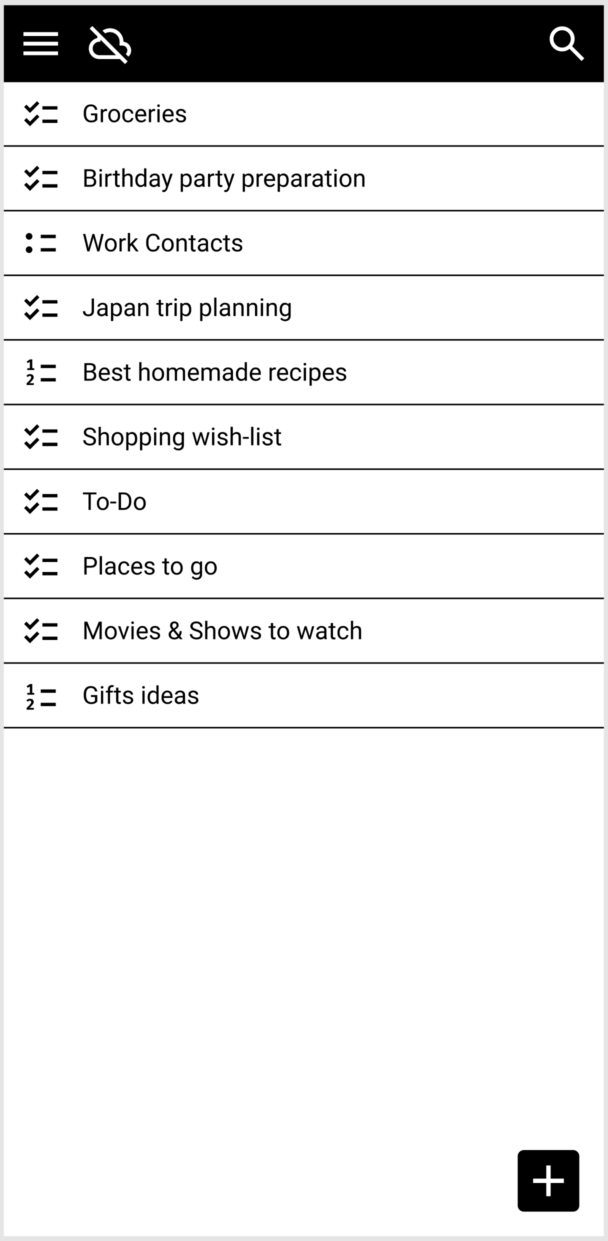
Finish **the** Figma design of the bachelor pro...
Studies > BA5 > Bachelor Project >

... **the** video editing of the family Easter reun...
Things to do > Family > Ideas >

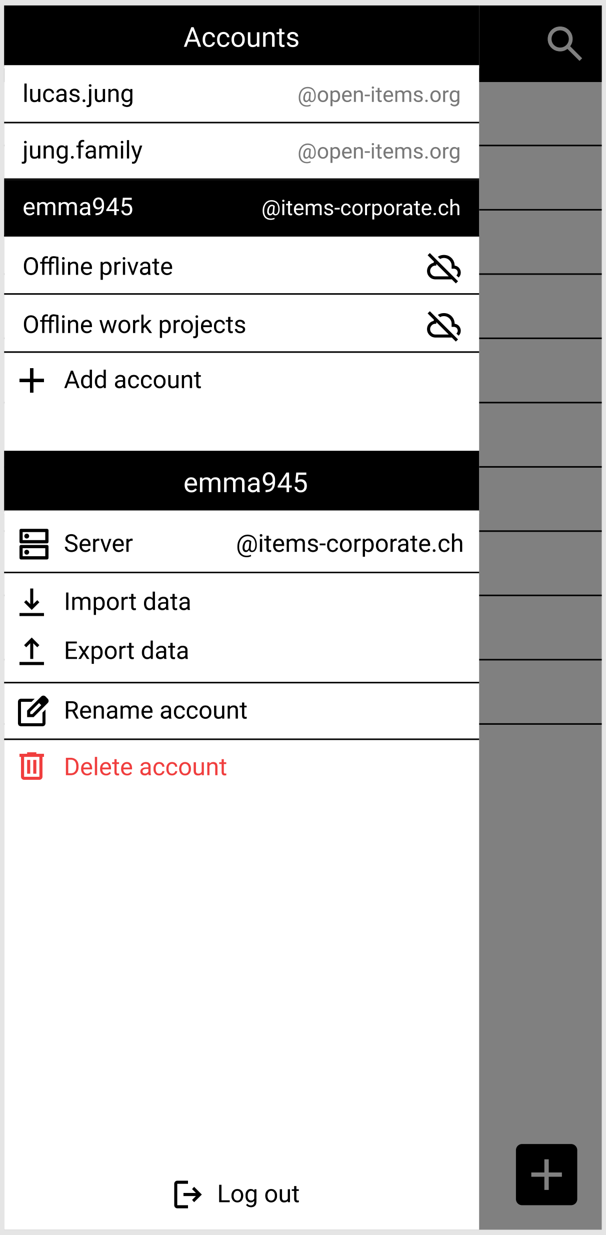
To work from **h**om**e**
Figma > Design > Examples >

Design 3

D3.1 – Lists page


Groceries
Birthday party preparation
Work Contacts
Japan trip planning
Best homemade recipes
Shopping wish-list
To-Do
Places to go
Movies & Shows to watch
Gifts ideas

D3.2 – Side bar


Accounts

lucas.jung	@open-items.org
jung.family	@open-items.org
emma945	@items-corporate.ch
Offline private	
Offline work projects	

Add account

emma945

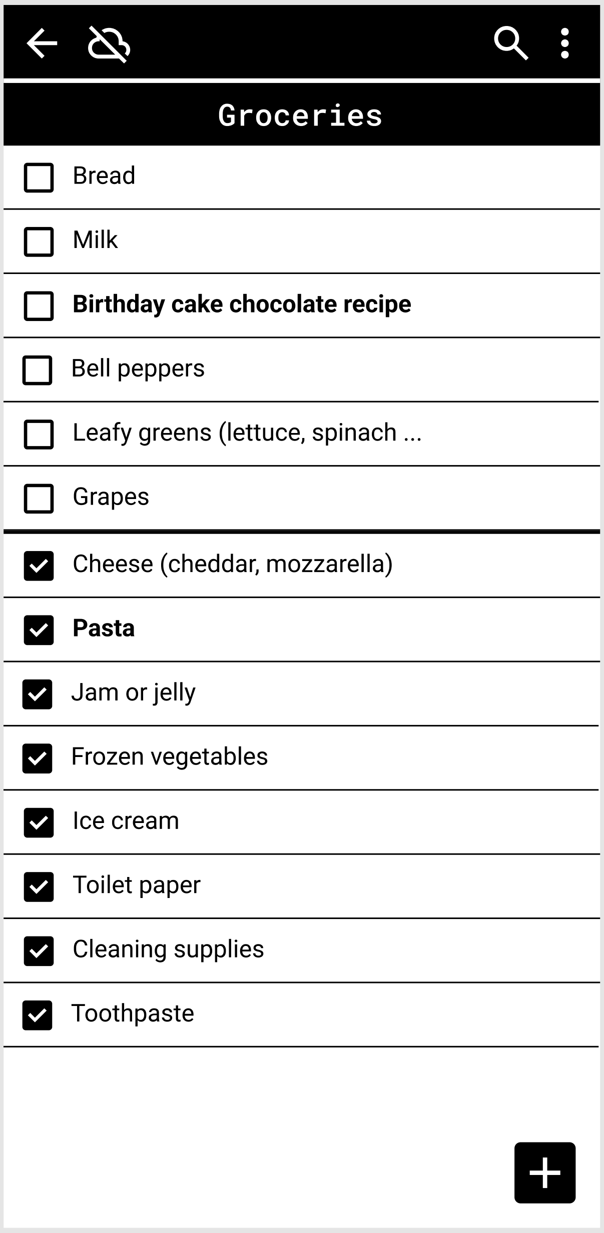
Server	@items-corporate.ch

Import data
Export data
Rename account
Delete account

Log out

Design 4

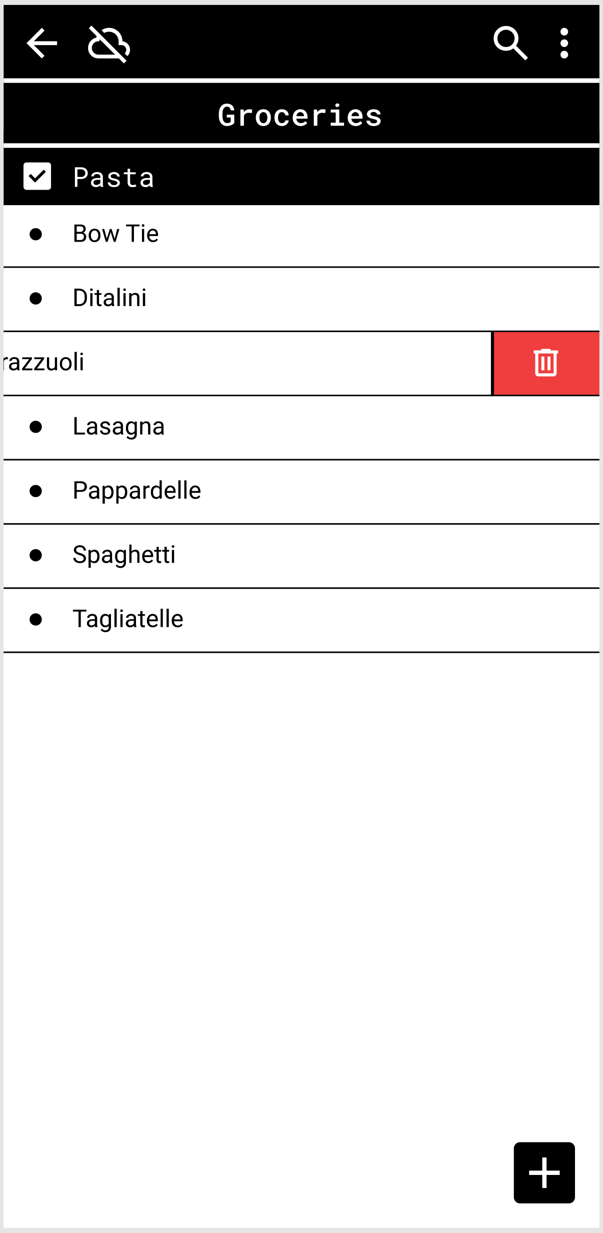
D4.1 – Items page



Groceries

- [ ] Bread
- [ ] Milk
- [ ] **Birthday cake chocolate recipe**
- [ ] Bell peppers
- [ ] Leafy greens (lettuce, spinach ...
- [ ] Grapes
- [x] Cheese (cheddar, mozzarella)
- [x] **Pasta**
- [x] Jam or jelly
- [x] Frozen vegetables
- [x] Ice cream
- [x] Toilet paper
- [x] Cleaning supplies
- [x] Toothpaste

D4.2 – Subitems page



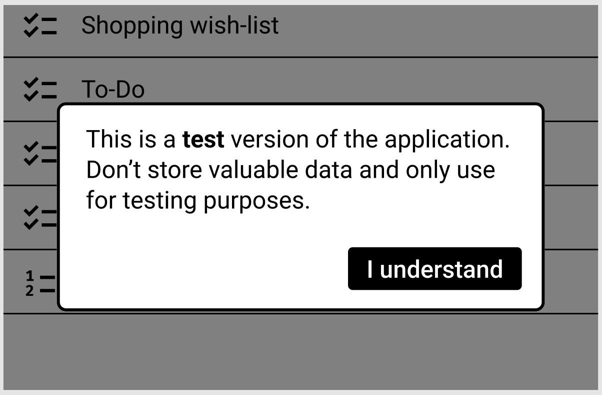
Groceries

☑ Pasta

- Bow Tie
- Ditalini
- razzuoli
- Lasagna
- Pappardelle
- Spaghetti
- Tagliatelle

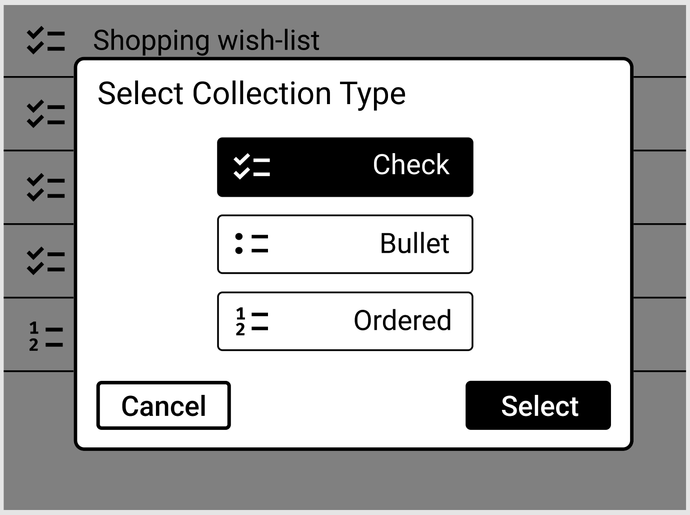
Design 5

D5.1 – Confirmation dialog

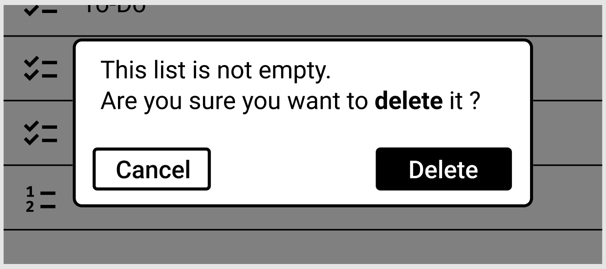


This is a **test** version of the application.
Don't store valuable data and only use for testing purposes.

D5.4 – Selection dialog

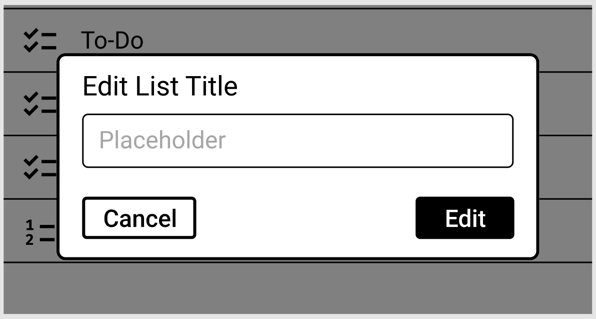


D5.2 – Cancel dialog

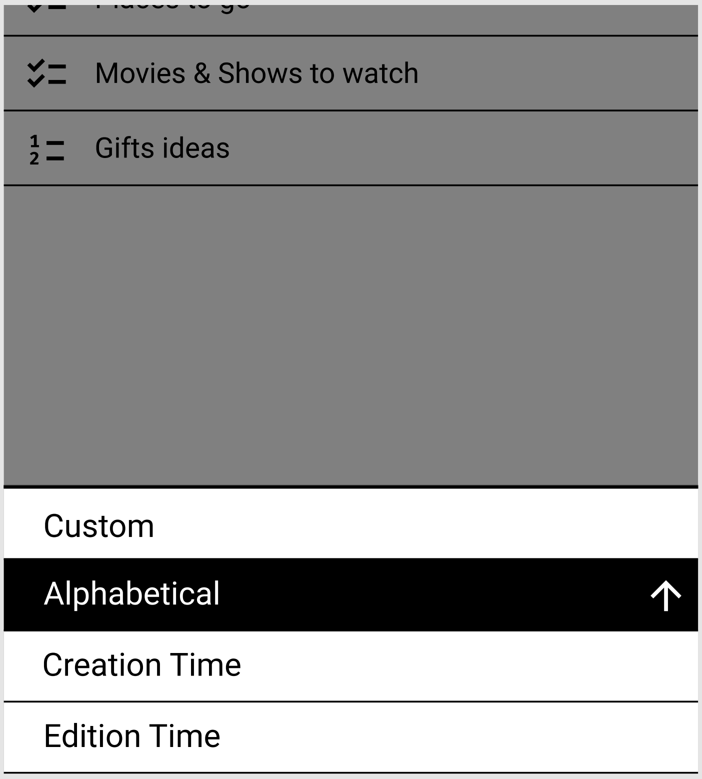


This list is not empty.
Are you sure you want to **delete** it ?

D5.3 – Text input dialog



D5.5 – Sort dialog


Movies & Shows to watch
Gifts ideas
Custom
Alphabetical
Creation Time
Edition Time