School of Electronic Engineering and Computer Science

Final Report

Programme of study:

Computer Science

Project Title:

TagRepo - A tool to assign and query tags on files in a filesystem

Supervisor:

Dr. Johan Pauwels

Student Name:

Chun Ho Wong

Final Year

Undergraduate Project 2022/23

Date: 29 April 2023

Abstract

In today's digital age, people are constantly creating and storing large amounts of data, ranging from personal photos and documents to work-related files. However, traditional file management systems, which rely on hierarchical folder structures, can be cumbersome and inefficient when dealing with a large number of files. This has led to an increased interest in alternative file management systems, such as tag-based approaches, which allow users to assign tags to their files and search for them based on those tags.

This report investigates and develops a tag-based file management system. The aim of this project is to provide a more efficient and flexible file management system compared to current hierarchical directory structures, enabling users to organise their files in a way that is suitable for them. The report identifies existing challenges related to personal information management (PIM) and file management, evaluates existing tagging software, and explores user needs in file management tasks. The report also discusses concepts related to PIM, investigates challenges related to file management, and proposes a tag-based system that allows users to manage and search files using tags. A user interface is developed that is easy to learn and efficient to use, and the effectiveness of the prototype system is evaluated through user testing. The report concludes by offering recommendations for further work.

Contents

Chapter 1: Introduction

1.1. Background

Computer users interact with digital files on a daily basis. This includes storing files, organising files, and retrieving files. The act of managing files and retrieving information is fundamental and necessary for general computer usage and many computer-based knowledge work. With modern systems and increased computer usage, the number of digital files on a user's system has increased significantly (Dinneen and Julien 2020).

The increase in the number of digital files on a user's system is related to the field of Personal Information Management (PIM). PIM is concerned with the management of digital files on the computer, and how people store, organise, and retrieve information to complete tasks. The increasing number of digital files on computer systems has made it more important than ever to investigate effective methods of managing digital files.

The directory tree structure is the de facto standard for data storage. Since its usage in early operating systems such as the OS/360 by IBM, most operating systems nowadays including Microsoft Windows, macOS and Linux distributions use directory trees as the data storage method. In such a system, filenames are used to uniquely identify files, and files can be placed within directories or folders.

The most common way of storing data on a computer is by using a directory tree structure. Organising files in a directory tree is analogous to sorting paper documents in real life - users can organise files by placing them into folders and further subfolders. While this is an intuitive way of organising files due to its similarities to organisation in real life, it is not an efficient way of organising and recalling files.

1.2. Problem Statement

Current file management systems are not efficient when it comes to managing a large number of files. Trying to search for a particular file can be a very time-consuming process. The main reason for this is that these systems are based on a hierarchical structure which makes it difficult to search for files based on their content (De Vocht et al. 2012). This problem is compounded when the users have to deal with a large number of files which can be difficult to organise consistently and structurally.

There are many methods available for users to search for files. This can include search functions, browsing through folders, or even looking through the recent documents list. However, these methods can be time-consuming and often give imprecise results that fail to find the file the user wants. This is a major problem for users who have to deal with large numbers of files.

Another major problem with current file management systems is that they are not flexible when it comes to organising files. This is because they are based on a hierarchical structure which can be inflexible and difficult to use. This can make it difficult for users to organise their files in a way that is suitable for them.

1.3. Aim

The project aims to investigate and develop a tag-based file manager. The system would allow users to assign tags to their files, and then search for those files based on those tags.

The aim of this project is to develop a file management system which is more efficient than current systems, and which enables users to organise their files in a way that is suitable for them.

Compared to a hierarchical directory structure, a tag-based approach would allow users to organise their files in a more flexible and efficient way. Users would be able to assign arbitrary tags to their files then search for those files using those tags. This lets users avoid the time-consuming task of classifying files and browsing folders, for example deciding which single folder to place a file into (Dinneen and Julien 2020). This would also enable users to search for files based on their content, and would make it easier for users to manage their files.

Users would still be able to organise their files into folders if they so choose. However, the system would not impose any strict structure on users and would allow them to store their files in any way they want. This is in contrast to existing tagging software which often override any existing hierarchical file structure and do not allow any hierarchical organisation.

The system should provide a search function which would enable users to search for files based on their tags. This would make it easier for users to find the files they are looking for. The system should also be able to handle a large number of files, such that the software remains performant in tagging and searching for large numbers of files.

The system should also be easy to use, such that users can use it without any prior knowledge of tagging or file management. This is in contrast to existing tagging software which often have a steep learning curve, and require users to have some knowledge of tagging before they can be used effectively.

1.4. Objectives

The objectives of the project are as follows:

- To identify existing challenges related to PIM and file management.
- To compare and evaluate existing tagging software available to users.
- To understand the needs of users when performing file management tasks.
- To gain an understanding of concepts related to PIM.
- To investigate the challenges related to file management.
- To develop a system to allow users to manage and search files using tags.
- To implement a user interface that is easy to learn and efficient to use.
- To evaluate the effectiveness of the prototype system through user testing.
- To evaluate the system against existing software systems.
- To conclude on the effectiveness of the system and to come up with recommendations for further work.

1.5. Research Questions

The research questions for this project are as follows:

- What do users need from a file management system in order to manage their files effectively?
- Can a tag-based approach be used to improve the efficiency of file management?

Chapter 2: Literature Review

2.1. Personal Information Management

Personal Information Management (PIM) is a field of research which is concerned with the management of digital files on the computer. This includes how people store, organise, and retrieve information to complete tasks (W. P. Jones and Teevan 2007). PIM is a relatively new field, and has only been studied extensively in the last few years. PIM is focused on how people organise, maintain and retrieve information, and on methods that can improve these tasks. This is not limited to digital files, but can also include paper documents for example.

One of the aims of PIM is for people to have the right information in the right place, in the right form and with enough completeness and quality to meet the current need (W. P. Jones and Teevan 2007). However this is not the case for most people. Often the necessary information may not be found by the user, or the information may arrive at an unsuitable time such that it cannot be used.

Jones mentions the idea of "keeping found things found", in which people store information in multiple location and multiple applications (W. Jones 2010). If this is performed inconsistently, the information people need is scattered widely, which makes it even more difficult to maintain and organise information. This is known as information fragmentation (W. P. Jones and Teevan 2007).

2.2. File Management

File management can be defined as the process of storing, retrieving and manipulating files on a computer system. This includes tasks such as creating, copying, moving and deleting files. File management is a fundamental task for users when using computers.

Most modern operating systems use directory trees to store data within files. A directory tree is a hierarchical structure in which each node represents either a file or folder. In this type of structure, filenames are used to uniquely identify each file within the system. Users can create directories or folders to group together related files.

While current methods for managing digital files are adequate for small workloads, they can become less effective with larger file collections. With larger collections, file hierarchies tend to have deeper structures (Henderson and Srinivasan 2009). This lead to file name duplication (Henderson 2011), and longer times spent to retrieve files (Bergman et al. 2010). Large hierarchies also require the user to make many navigational decisions with many subdirectories per directory (Hicks et al. 2008).

There are many methods that can be used to help deal with large file collections. These include using automated organisational methods, and providing users with better tools for managing their files. Automated methods can be used to help users organise their files into a more manageable structure. This may include grouping files based on content or context using artificial intelligence algorithms.

2.3. Tagging

A tag is a label or keyword that can be attached to an item. One benefit of using tags to categorise items is that items can be easily found and classified. This can be useful for many tasks, such as information retrieval and content organisation.

One benefit of tagging over hierarchical systems is that it is more flexible. This is because tags can be arbitrarily named depending on the user's needs, and then assigned to multiple items at once. Items do not have to fit into predefined categories.

Tagging rose into popularity through its usage in many websites in Web 2.0 (Smith 2007). Tagging is now widely-used, and can be found in many social media websites and online services. One example of this is category tags in news websites and blogs.

Tagging has also been used to help users organise their files on the computer. There are some tools available that allow users to tag their files, and then search for them using those tags. This provides a more flexible way of organising files compared to traditional hierarchical directory structures. It also makes it easier for users to find the files they are looking for since they can search using any combination of tags.

Tagging has many advantages over traditional methods for managing digital files. The most significant advantage is that it enables users to search for files based on their content, rather than just their filename or location. This makes it much easier for users to find the files they are looking for. Another advantage of tagging is that it does not require any specific structure or organisation, which may make it easier for users to manage their files.

There are also some disadvantages associated with tagging. One of the main disadvantages is that manual tagging can be a time-consuming process, particularly if the user has a large number of files. Another disadvantage is that automatic tagging systems are not always accurate, and can often assign incorrect tags to files.

2.4. Existing Programs Review

There are many software solutions for using tags for file management. Some of them are standalone applications, while some may be integrated into the operating system.

2.4.1. Windows' Native Tagging System

All Windows versions since Windows 7 has built-in tagging functionality. However this functionality is well-hidden and difficult to access. On files that support tags, a field for a list of tags is available on the "Properties > Details" popup panel.

Windows also provides a method for searching for files based on tags. On the search bar in Windows Explorer, the syntax `tag:<tagname>` can be used to search for files containing a specific tag.

The main advantage of this solution is that it is integrated into Windows, requiring no additional software. This makes it easy for users to get started with the system, and makes it easy to use since users are likely already familiar with File Explorer and Windows Search.

The main issue with Windows' native tagging system is that not all filetypes support tags. For example, tags cannot be applied to PNG image files and various audio formats including WAV and MP3. This is a significant limitation given that PNG, WAV, MP3 and other formats are commonly-used, users are likely to have many files of these types.

Another issue with Windows' tagging system is that users can only assign tags to each file at a time. It is not possible to assign a single tag to multiple files at the same time. With a large number of files, tagging each file will become a very time-consuming process.

One more issue is the speed of Windows' search functionality. Windows does not cache or index the tags contained in each file. This means that the system needs to scan each file for the specific tag the user requested when performing a search. The time required to search for files grows proportionally with the number of files the user has.

2.4.2. TagSpaces

TagSpaces is an open source application which runs on Windows, macOS and Linux.

It provides basic features such as assigning tags to files, and searching files based on tags. TagSpaces tags files by directly renaming the file with a prefix containing the tags to be assigned, wrapped with square brackets.

One benefit of this approach is that it is able to track file movement. Files that are moved using the file default program can still be found by TagSpaces since the tags are part of the filename. This also means that a file's tags are viewable using any file manager by simply viewing the filename.

One issue with TagSpaces is that renaming files can break links between applications and files. For files that act as dependencies for other application, this method cannot be used. For example, an audio file `cat.wav` may get renamed to `[animal recorded short purr] cat.wav`. Any third-party applications that are linked to the `cat.wav` file will now be unable to find the file and give an error.

Another issue with TagSpaces is that the filenames can get very long as users assign more tags to files. With a large number of tags on a file, the original filename will be prefixed by a large amount of text, making it difficult to view the original file name using file managers. For example, a file name like `[photo 2019 holiday uk london beach sunny clouds sea me dan jason burger beer sunglasses pub] IMG_7690.png` would easily get truncated in file managers, the user will not be able to view the full file name without resizing the entire window.

2.4.3. Tabbles

Tabbles is a desktop application that only runs on Windows systems. It comes in several variants: individual, cloud, and LAN. The different versions differ in how tags are shared between users on the same network or users connected to the cloud.

Tabbles tags files using a backing database, separate from the files being tagged. When the user moves a file to a different folder, a background service is used to detect such file movements and update the database to use the new path.

It provides many features. It allows grouping tags into a hierarchy, creating auto-tag rules, and tagging of entire folders for example.

One advantage of Tabbles is its auto-tagging feature. This feature allows for automatic assignment of tags to files based on specific file conditions. This helps the user save time when organising and managing files by automating the task of tag-assignment.

One issue with the software is that tags are not transferrable between computers. Tags are stored in a database that is associated with the system. This makes it very difficult to transfer tags if the user needs to upgrade their operating system and reinstall their system, or if the user wishes to upload their files to cloud backup.

From personal testing, another one of the issues is its slow performance. The application takes a long time to execute searches. When loading a large list of files, it slows down significantly as it scans each file to generate image previews.

2.5. Programming Tools and Frameworks

For the development of this project, a variety of tools and frameworks will be used. In this section, I will explain my decisions when it comes to deciding what toolset to use.

2.5.1. Rust

Rust is a programming language designed by Graydon Hoare. It is designed to be memory-safe while still being performant (Balasubramanian et al. 2017). This is enforced by the Rust compiler, which checks memory-safety rules during compilation.

Rust is increasingly adopted by companies and projects such as Microsoft, Amazon, Cloudflare, and Facebook (Davison 2022). It has also been integrated into the Linux kernel (Proven 2022).

Though I have years of experience in high-level languages such as Python, Ruby, Javascript, I have very limited experience with low-level languages like C++ and Rust. However, since Rust is being increasingly adopted by industry, as well as its memory-safety features, performance and industry support which make it a good choice for this project, it was an obvious choice for me when deciding upon a programming language for development. I take this opportunity to learn Rust as it will also help with my career.

2.5.2. Tauri

Tauri is a Rust framework that allows developing desktop applications using web technologies like HTML, CSS, JavaScript for the frontend, while using Rust for the backend (Tauri Contributors 2022).

Rust is a relatively new language compared to other languages. It has many user interface libraries, however most of them are in the early stages of development and not yet fully-featured (Bensing 2019). Among these, Tauri stands out for its support of many platforms and its polished feature set.

Tauri uses web technologies for implementing user interfaces. This allows developers to use widgets and components from frontend frameworks such as Vue and React - developers are not limited to a small subset of widgets implemented by the library author. Tauri can also take advantage of existing tools and libraries from the web development ecosystem - this provides a more complete solution.

The Rust backend can be used to access the file system and other desktop-specific features. This provides a more complete solution compared to using web technologies alone, for example building a browser-based file manager.

In preparation for this project, I developed a small program using Rust, Tauri and Vue to learn more about these technologies and gain experience in software development. The

program was successfully completed and I was able to gain a better understanding of Rust, as well as the features made available by the libraries Tauri and Vue.

2.5.3. Vue

Vue is a JavaScript frontend framework developed by Evan You. It is designed to be flexible and easy-to-use, enabling the efficient development of web applications.

Vue combines aspects from modern frontend frameworks such as Angular.js and React. It generates a virtual DOM and renders this to the real DOM. It has many features such as two-way binding, reactivity, computed properties, single file components (SFC).

Vue has developed into one of the most popular frontend JavaScript frameworks. In 2022 it reached 3rd place in terms of popularity, below React and Angular and above Next.js and Svelte (Stack Overflow 2022).

I chose Vue as the frontend framework because it has good documentation and many third-party libraries, and is a well-supported framework used by companies such as Facebook, Netflix, and Adobe.

I have some experience using Vue, but only in a limited capacity - I have experience with this framework through one of the university modules I am studying. I decided on this opportunity both improve my skills with the framework as well as developing new skills related to desktop application development.

2.5.4. SQLite

SQLite is a file-based relational database management system developed by D. Richard Hipp. It is designed to be lightweight, making it a good choice for small desktop applications since it does not require any external dependencies such as running an entire SQL server application in the background.

SQLite has basic features including transactions, triggers, views and foreign key support so that queries can be easily made using JOIN clauses. This gives tagging software access to many useful query functionalities without the overhead of running a separate process for hosting a database connection.

SQLite has a wide range of applications, including being used as an embedded database in Firefox and Android. It has good documentation, many client libraries and an active community of users.

I chose SQLite because I have plenty of experience using databases from my own personal use and through a part-time developer role. I would be able to implement a tagging system using SQLite and Rust without much difficulty.

Chapter 3: Requirements Analysis

The following requirements were obtained through a review of existing tagging software and other file management systems, and through online sources such as blogs and discussion forums.

3.1. Functional Requirements

1. The system should allow users to tag files with arbitrary tags.
2. The system should provide a search function which would enable users to search for files based on their tags. This would make it easier for users to find the files they are looking for.
3. The system should preserve any existing directory tree structures, avoiding automatic renaming or moving of files.
4. The system should be able to handle a large number of files, such that the software remains performant in tagging and searching for large numbers of files.
5. The system should be easy to use, such that users can use it without any prior knowledge of tagging or file management.
6. The system should be able to tag any type of file, including but not limited to images, videos, text files, PDFs.
7. The system should be cross-platform and work on Windows, macOS and Linux systems.
8. The system should have a graphical user interface for tagging files and searching for files using tags.
9. The system should allow tags to be assigned to multiple files at once, rather than individually tagging each file one-by-one. This would reduce the amount of time needed for tag assignment, particularly for large numbers of files.

The following requirements are optional extensions to the project and not required for the base functionality.

8. The system should be able to handle transferring of tags between different devices or systems. This would allow users to preserve their tags when transferring across multiple systems, and would also allow files to be stored on cloud backup services such as Dropbox or Google Drive.
9. The system should display previews of each file, for example preview of audio files, thumbnails for image files, and extension icons for unrecognised file types.

3.2. Non-functional Requirements

1. The system should be able to execute and complete searches within 1 seconds for large file collections up to 10000 files.
2. The system should have a response time of less than 500ms for all file operations such as opening, renaming and tagging files.
3. The source code should be well-documented such that it is easy to understand and maintain.
4. All user input should be validated both on the frontend and backend to ensure data integrity and security.
5. The application's graphical user interface should be consistent across all platforms, providing a familiar experience for users regardless of their platform choice.

Chapter 4: Design

4.1. Components

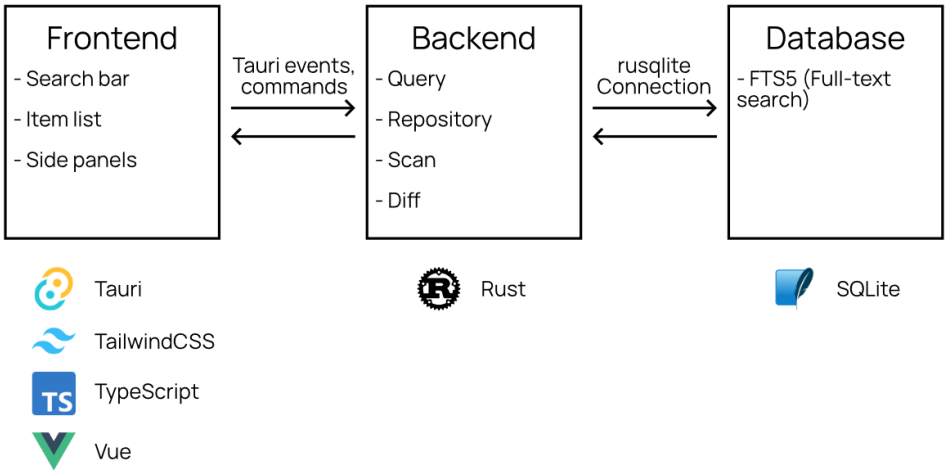


Figure 1: Components of the application

The application is made up of several components. The Rust backend is responsible for file operations, database access and cross-platform compatibility. The Vue frontend is responsible for the graphical user interface. The SQLite database stores tags assigned to files by users. These components are explained in more detail below.

4.1.1. Rust Backend

The Rust backend is responsible for file operations, database access and cross-platform compatibility. The backend is implemented using the Rust programming language and the Tauri framework. It is divided into many modules for code organisation.

The query module is responsible for parsing user queries and converting them into SQL queries. This is further discussed in Section 5.1.

The repository module is responsible for interacting with the SQLite database. It contains the repository model definition, as well as functions for managing connections and high-level abstractions for managing database contents.

The scan and diff modules are responsible for listing files in filesystem. Since the application may be opened and closed frequently, it needs to be able to keep track of the files in the file system after the application has been closed. The application achieves this through a syncing operation on application startup, it scans the folder for files using the scan module and determines which files are created, deleted, or removed since the previous session using the diff module.

The watch module is responsible for real-time tracking of files in a folder. The module is used to track changes to a folder while the application is open, allowing the user to modify files freely while the application is open and keep any tags on files intact. This is further discussed on Section 5.4.

4.1.2. Vue Frontend

The Vue frontend is responsible for the graphical user interface. It is implemented using the Vue JavaScript framework, TailwindCSS for styling, and Vite for building and optimising the frontend.

The frontend is created using the build tool Vite, it operates on a single HTML file and recursively checks any linked stylesheets and JavaScript files. This includes the Vue frontend framework and the TailwindCSS library.

The frontend uses TailwindCSS for consistent styling across all HTML renderers on different platforms. TailwindCSS is built on the modern-normalize framework, which aims to normalise styles across various browsers on different operating systems (Cornes 2020). This allows the application to retain the same appearance across different operating systems.

4.1.3. SQLite Database

The SQLite database stores tags assigned to files by users. The database is implemented using the SQLite 3 library and its full-text search extension. The SQL schema is included as an appendix at Section 8.1.

The “items” table contains several columns - an autoincrementing ID, the path of the file, a list of tags, and a list of metadata tags for query generation. The path column is used to store the relative path of the file to the root of the repository. The tag name column is used to store a list of tags assigned to the file. The metadata tags are used for the query module to construct accurate SQL statements, this is further discussed in Section 5.1.

The “items_fts” table is a virtual table using SQLite’s FTS5 full-text search extension. It allows quick indexing and querying of any text-based columns. The virtual table indexes and searches the tag column on the “items” table.

The database is accessed from Rust using the `rusqlite` crate. It provides functions for opening and closing a database connection, executing SQL queries, as well as registering custom functions provided by my own code.

4.1.4. User Interface Design

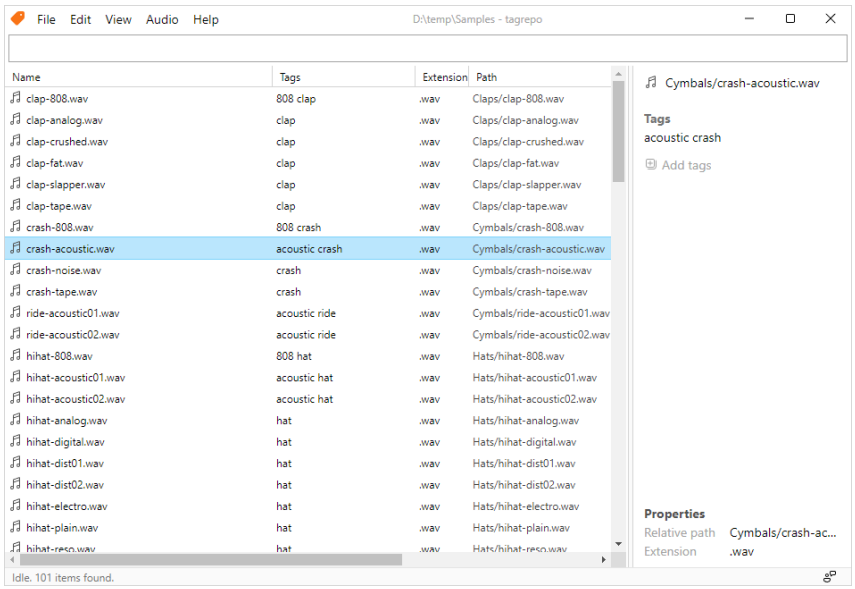


Figure 2: Current user interface, 15 April 2023.

The user interface is designed to be simple and easy-to-use. It consists of a toolbar, a search bar, the main content area, and a properties panel. The toolbar contains common actions and options for the application. The search bar allows the user to input arbitrary queries. The main content area contains a list of files and information about each file. The properties panel displays tags assigned to the selected files and allows the user to assign new tags to them.

The application is designed to be used with a mouse and keyboard. The toolbar buttons and main content list can be navigated using the mouse. Keyboard shortcuts are provided for convenience, such as pressing Enter on the query bar to switch focus to the file list, pressing arrow keys in the file list to navigate to the next / previous item.

The sidebar is used for tagging files selected in the main content area. Users can edit the list of tags for the selected file. If multiple files are selected, the sidebar shows common tags between all selected files, any new tags will be added to all selected files.

The user interface is designed to be consistent across all platforms. The application should look and feel the same on Windows, macOS and Linux systems.

4.2. Searching for Items

The application is designed to be a file manager. It should allow users to search for files using tags, but at the same time should not disregard the actual file structure of the underlying file tree. Therefore, users should be able to search for files not only using tags, but also using file attributes such as the file path.

The application uses SQLite's FTS5 full-text search extension to facilitate searching files by tag. When combined with other SQL conditions, the following is one example of such query:

```
SELECT i.id, i.path, i.tags, i.meta_tags
FROM items i
INNER JOIN
    tag_query tq ON tq.id = i.id
WHERE
    tag_query = '(a NOT b)'
    AND i.path LIKE 'samples/%';
```

The query searches for items that are under the path `samples/` and have the tag `"a"` but don't have the tag `"b"`.

To keep the software accessible and usable to as many users as possible, the software should not expect users to enter SQL queries directly since most computer users do not have experience in SQL. For more complex queries, the resulting SQL statement would become difficult to write and to understand.

As such, the application provides a plain-text query language that gets converted into SQL behind the scenes. The above query may be expressed in this new query language as follows:

```
a -b in:samples/
```

Terms such as `a` and `b` are treated as tag names, while terms with the prefix `in:` are file path queries. Terms that do not have any operators between them are implicitly joined with an

`AND` group. Additional operators such as `|` and `-` allow grouping terms using boolean `OR` and `NOT` queries.

The application should convert the above plain-text query into the same SQL statement above when executing the query.

Chapter 5: Implementation

5.1. Parsing Plain Text Queries

To support the plain-text query language described in Section 4.2, the application needs to implement a compiler that translates from the plain-text query language to SQL.

This is one of the major challenges when implementing the software. In order to support arbitrarily-complicated queries, the compiler must be able to handle any combination of search operators and many edge cases.

Since the application fetches data from the same tables for every query, the `SELECT` and `JOIN` clauses of the SQL statement stay constant - I only need to consider the `WHERE` clause of the SQL statement when converting from a plain-text query. The following is the SQL template used in the compiler:

```
SELECT i.id, i.path, i.tags, i.meta_tags
FROM items i
INNER JOIN
    tag_query tq ON tq.id = i.id
WHERE
    :converted_plain_text_query
```

When given a plain-text query, the application converts it into a SQL `WHERE` clause, then inserts it into the template. However, due to differences between the plain-text and SQL languages, there is no straight-forward way to convert plain-text queries to SQL `WHERE` clauses. I will explain those differences by examining a few cases.

5.1.1. Query Edge Cases

5.1.1.1. Case 1: FTS5-only queries

```
-- The plain-text query:
-- "Item with tags a and b, or items with tag c without tag d"
a b | c -d

-- The SQL WHERE clause
WHERE tag_query = '(a AND b) OR (c NOT d)'
```

In the simplest case, the plain-text query only contains tag queries.

Terms prefixed with a minus symbol (`-`) are "NOT" clauses and have the highest precedence; terms next to each other without operators are "AND" clauses; terms separated with a bar (`|`) are "OR" clauses and have the lowest precedence.

In terms of SQL, the plain-text query only queries using the FTS5 extension without accessing any other columns. In this case, the resulting WHERE clause will only contain one expression - the FTS5 expression. Conversion from plain-text to FTS5 is relatively simple, except for an edge case:

```
-- The plain-text query:
-- "Item with tags a and b, or items without tag d"
a b | -d

-- The SQL WHERE clause
WHERE tag_query = '(a AND b) OR (meta_tags:all NOT d)'
```

The FTS5 extension treats `NOT` as a binary operator, not a unary operator. In other words, the `NOT` operator computes the set difference of two conditions. It cannot search for the complement of a single condition. Often, it can be useful to search for the complement of a single tag, for example searching for all items that don't have the tag `old`. However, it is impossible to represent complements in FTS5.

To circumvent this issue, I added a new column to the `items(id, path, tags)` table called `meta_tags`. The `meta_tags` column contains the string `"all"` by default, so all items in the table will gain a new `all` tag. I can then specify `meta_tags:all` in the FTS5 query to search for *"all items that have the tag 'all' in the 'meta_tags' column"*. I finally use `meta_tags:all` as the first operand in the `NOT` operator to compute the set difference between all items and the given query, effectively implementing unary negation of a single term.

5.1.1.2. Case 2: FTS5 and SQL queries without OR operands

```
-- The plain-text query:
-- "Item in the folder 'my_folder' with tags a and b"
a b in:my_folder

-- The SQL WHERE clause
WHERE tag_query = 'a AND b' AND i.path LIKE 'my\_folder%' ESCAPE '\'
```

For clarity, "FTS terms/expressions" will refer to tag searches like "a b", while "SQL terms/expressions" will refer to non-tag searches like "in:my_folder".

In this case, the WHERE clause requires multiple expressions in addition to the FTS expression. For each SQL term in the query, the WHERE clause must include a separate SQL expression for the term. All the rules and edge cases from Section 5.1.1.1 apply here. This case is similar to case 1 and is simple to handle.

5.1.1.3. Case 3: FTS5 and SQL queries with OR operands

```
-- The plain-text query:
-- "Either i) item in the folder 'my_folder' with tags a and b, or
--         ii) item in the folder 'other_folder' without the tag d
a b in:my_folder | -d in:other_folder

-- Incorrect SQL clause
-- This will not behave as expected
WHERE (
  (tag_query = 'a AND b'
    AND i.path LIKE 'my\_folder%' ESCAPE '\\')
  OR
  (tag_query = 'meta_tags:all NOT d'
    AND i.path LIKE 'other\_folder%' ESCAPE '\\')
)

-- The working SQL WHERE clause
WHERE (
  (i.id IN (SELECT id FROM tag_query('a AND b'))
    AND i.path LIKE 'my\_folder%' ESCAPE '\\')
  OR
  (i.id IN (SELECT id FROM tag_query('meta_tags:all NOT d'))
    AND i.path LIKE 'other\_folder%' ESCAPE '\\')
)
```

When the plain-text query contains FTS5 terms and SQL terms joined together with at least one OR operator, the resulting SQL statement must make use of subqueries.

This is due to two limitations of SQLite's FTS5 extension. First, it only supports at most one FTS5 expression in the WHERE clause. Attempting to query using multiple FTS expressions will result in zero rows being returned. Second, it does not handle OR groups that contain a FTS5 expression correctly. A query such as *"a b | in:my_folder"* will have rows missing from the output.

To overcome this, I replace all FTS expressions with a subquery that contains the required FTS expression. In effect, this makes the statement behave as expected.

5.1.2. Compiler Implementation

The compiler is implemented using two files, `parser.rs` and `convert.rs`.

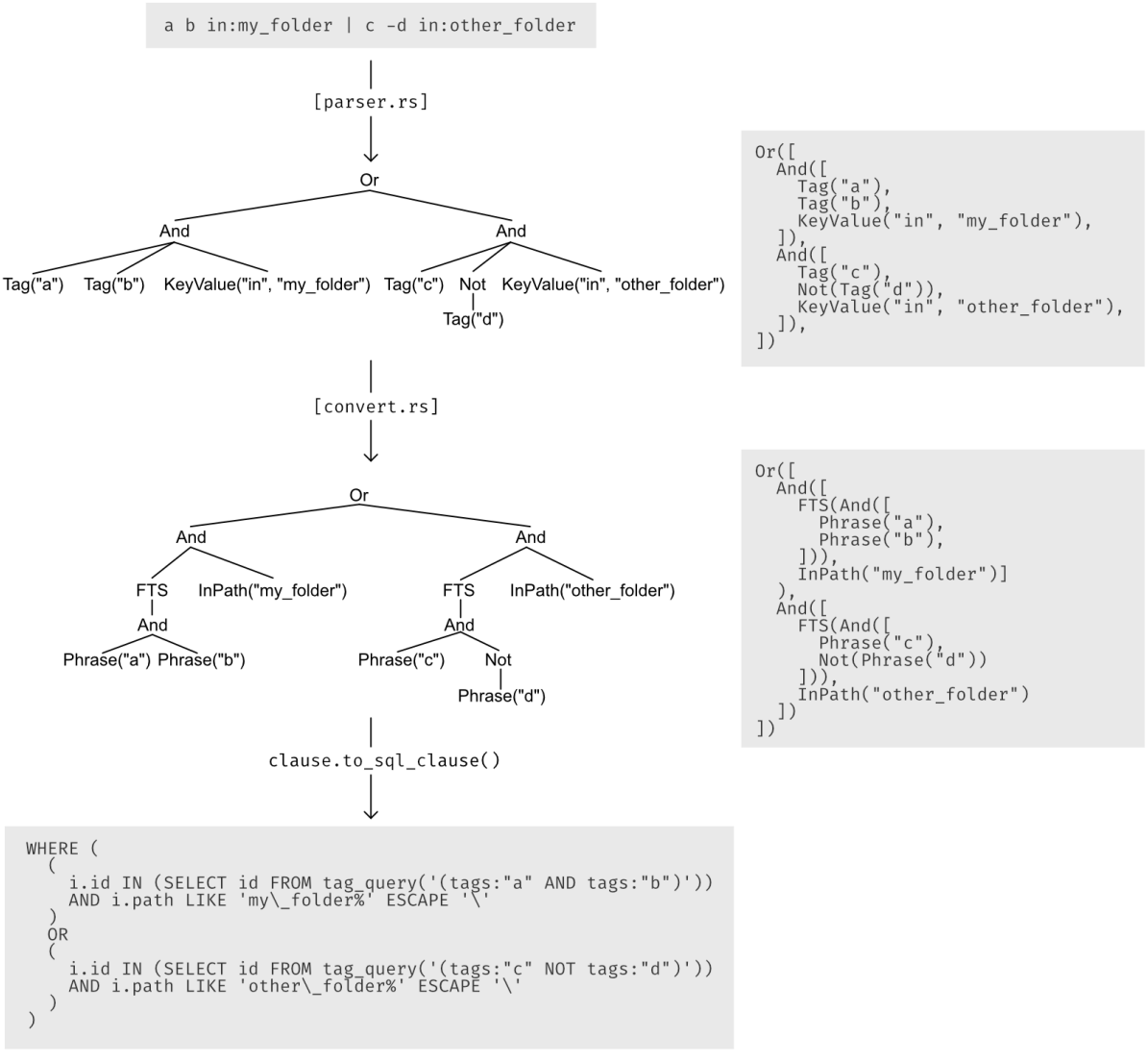


Figure 3: Diagram showing the process of converting a plain-text query to a SQL statement

The parser is implemented in `parser.rs` using the "nom" crate (Couprie 2021). Its purpose is to validate and parse a plain-text query into a parse tree.

The purpose of `convert.rs` is to convert the parse tree into an abstract syntax tree (AST) (Thain 2019). In this stage, it combines any FTS terms on the same level into a single `FTS()` object that represents a single FTS expression in the output SQL statement.

The conversion from the AST to SQL code is also handled by `convert.rs`. This is done by calling the `to_sql_clause(&self)` method on the base of the tree, which then recursively calls `to_sql_subclause(&self, is_root: bool)` on its child nodes.

Finally, the compiler inserts the SQL clause into the SQL template to obtain the final SQL statement. The statement can then be used to query items in the database.

The final number of lines of code in the compiler module was 1149 lines.

5.2. File Paths and Operating Systems

One of the requirements of the software is cross-platform compatibility: the software should run on all major operating systems, and the produced database must be transferable between operating systems. However, in terms of file path handling, Windows and Unix-like operating systems (including macOS and Linux) have drastic differences.

The first major difference is path separators. Windows defaults to backslashes (`\`) to separate components in a path, but also recognises forward slashes (`/`) as valid path separators in command contexts. Unix systems use forward slashes to separate path components, whereas backslashes, while heavily discouraged, are allowed as normal characters in filenames.

The second major difference is partitions. A Windows system divides disk space into partitions, each of which has a unique drive letter such as `C:\` and `D:\`. Windows absolute file paths will always contain drive letters, for example "`D:\Audio Samples`". Unix systems do not use partitions, on such systems file paths always begin with a forward slash, for example `/home/james/audio`.

These differences present several issues when storing paths in the database. If the application directly stored the absolute paths of files in the database, this would mean storing either Windows-specific paths or Unix-specific paths in the database. The database would become operating system-specific and is no longer cross-platform.

The issue of path separators is easy to solve - we only use forward slashes as separators since that is recognised as a valid separator on all operating systems. On Unix, this requires no changes to the paths and thus has no effect. On Windows, this can be implemented with a simple string substitution.

The issue of drive letters is more difficult to handle. If we wish to store absolute paths in the database, there is no way to create absolute paths that are valid on both Windows and Unix systems, since Windows drive letters are invalid on Unix systems.

Thus, the software must be limited to relative paths for cross-platform compatibility. This means the software must choose a location as the root directory, and store paths relative to the root directory. In the software, this is abstracted with the concept of "repositories", which is a folder where tags are managed and stored. This ensures that all paths can be stored as relative paths relative to the root of the folder. If the software doesn't track files outside the folder, this also has the added benefit of lower performance overhead as the software does not need to keep track of files in the entire system, including system files that

the user may not want to tag. This also allows the user to create several isolated folders of tags, each folder dedicated for a different purpose and containing different tags.

5.3. Optimising the Item List

One of the requirements for this project is for the software to remain responsive when handling large file collections. The main item list of the user interface is a major component of the software that the user will interact with frequently, thus it is important to ensure that the item list is sufficiently responsive.

A naive implementation of the item list may be as follows: When the user enters a new search query, the frontend sends the query to the Rust backend. The backend queries the database for a list of items, then returns a list of each item and any associated information such as item paths to the frontend. Upon receiving the list, the item list renders every item in the received list using data that is stored in the list.

The above implementation works well for small file collections, but does not scale with larger collections.

As the length of the list increases, the data needed to be transferred from the backend to the frontend increases very quickly, and requires a longer time to complete. This causes a momentary freeze in the user interface lasting 1 to 5 seconds or more depending on the size of the list.

After the transfer, the item list then renders every item in the list. In a DOM-based (Document Object Model) user interface library like Vue, this means creating DOM nodes for every item in the list. This can quickly become expensive, especially for larger lists. This is reflected in the user interface as stuttering when the user scrolls the list.

The software implements two optimisations to address these issues.

5.3.1. Virtual Item List

The first optimisation is a virtual item list. A virtual list is a technique for rendering only the items visible on-screen in a scrollable list, while skipping the rendering of all other items to improve performance.

In Vue, a virtual item list massively reduces the number of DOM nodes generated for a given item list. Since the number of items visible depends on the size of the application window, the number of rendered items remain constant as the length of the item list increases. This way, the item list only generates a fixed number of elements, and the user can still interact with all items as if they were all present on the page.

The full code for the item list is included as an appendix at Section 8.4.

In my implementation of the virtual list, I added a buffer zone to pre-render items that are slightly off-screen. The reason for this buffer zone will be explained in the next section.

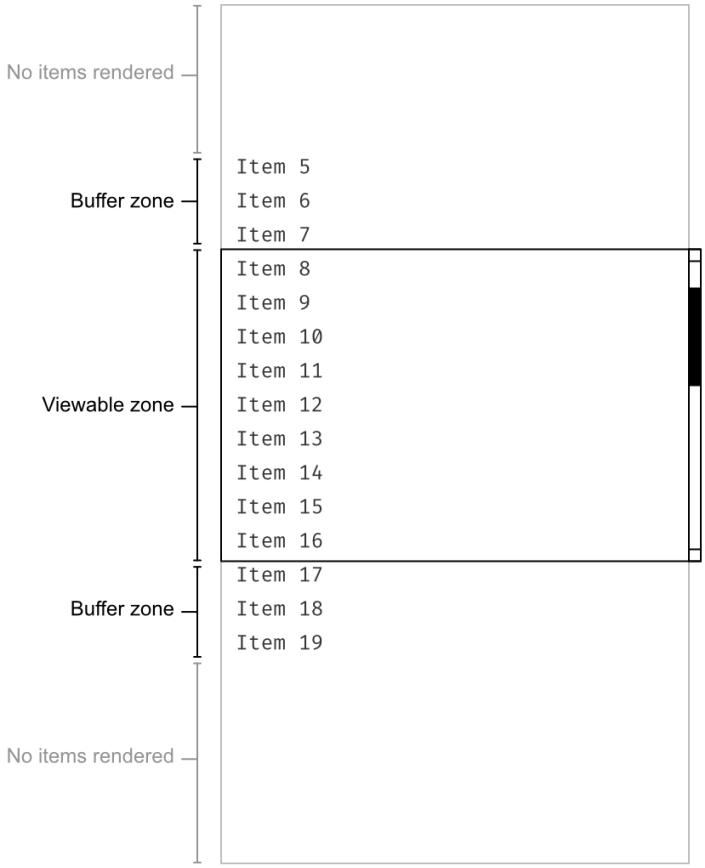


Figure 4: Diagram showing the different regions involved in a virtualised list.

5.3.2. Limiting the Data Transferred

To reduce the amount of data transferred from the backend to the frontend during the initial load, we observe that not all of the data is useful when the list first loads. In a large list of items, only the paths and tags of the first few items will be visible, whereas the path and tags of all other items are not visible until scrolled to. Thus it is unnecessary to transfer the paths and tags of all items for the initial load, the only necessary information is the item ID, which uniquely identifies the item in the database.

In the software implementation, the virtual list initially only has access to a list of item IDs. If it wishes to get more information about an item to render it, it must make another request to the backend to fetch the item data. The fetching of items is not instantaneous and takes about 5-10ms for each item.

A further optimisation is to cache the data obtained from the backend. I implemented an item data cache that stores any retrieved items until the next query. When the virtual list attempts to fetch data from the backend, it now checks the item cache for existing data before attempting the fetch.

One downside of this approach is that the rendering of items is no longer instantaneous, since it needs to wait for the backend to respond with the relevant data before being able to render the item. To address this, the virtual list includes a buffer zone to pre-render items that are slightly off-screen. This ensures that items have enough time to render before being scrolled into view.

5.4. Extension: Directory watcher

The ability to track file movement is essential to this application. When implemented, it allows the application to preserve tags on a file when the user moves a file to a new location. However, implementing this on the Windows operating system presents a major challenge.

5.4.1. Detecting File Movement on Windows

Windows provides a native API to watch a directory for changes. However, one major issue with this API is that it is unable to detect file movement. The API can emit events for file creation, removal, and in-place renames. However, file movement from one directory to another is simply detected as a pair of file-create and file-delete events.

This prevents it from being able to be used directly as the watcher for the application. When used as the watcher for the application, the application is unable to preserve tags on a file when the user moves a file to a different folder. This is a major issue for many existing tag-based file managers as well - they often lack the ability to track file movement, have unorthodox solutions that affect user usability, or make it the user's responsibility to manually update tags after file movement.

To solve this issue, I implemented an asynchronous event handler that takes Windows' native events as input, and automatically resolves file movement by checking the file paths of the received events.

This watcher is only used on Windows systems. On other systems, their respective default watchers are used instead with conditional compilation.

5.4.2. Final Implementation

```
Watcher
  |
  v
Handler
  |
  v
Output
```

Listing 1: Diagram showing the final iteration of the watcher

I observe that moved files retain the same file name, even if they are moved to a different directory. The solution I implemented makes the following assumption: A delete event, followed shortly by a create event with the same file name is caused by a file rename event.

The final implementation used two components: the watcher, and the handler.

The watcher is an instance of a `Watcher` from the `notify` file watching library. It watches a directory and emits events generated by the underlying operating system (on Windows, the events come from the Windows API). In Windows, file movement is treated as a pair of create and delete events.

The handler is an asynchronous infinite loop, iterating through all events received from the watcher. It is responsible for determining whether a received event can be used as-is, or whether the event must undergo further processing and be deferred. The following sections describe the algorithm used to process events.

5.4.2.1. Handler Algorithm

In Windows' file system, all events except create and delete events can be used as-is, these events will be sent directly to the output. Create and delete events may correspond to a single rename event, as such they are handled differently from other events.

When a delete event is received, it may correspond to either a file movement or a file deletion. The algorithm stores this event in a list, and adds to the event an expiry date. The reason behind the expiry date will be explained in the next section. If the path is determined to be a rename event before the expiry time, then the algorithm sends a rename event. Otherwise, the path expires and is treated as a delete event, which the algorithm sends to the output.

When a create event is received, it may correspond to either a file movement or a file creation. The algorithm goes through the list of recently-deleted paths and checks if any path has the same file name as this create event. If a matching file name is found, it marks the path as a rename event, then sends a rename event to the output. Otherwise, the algorithm returns the create event as-is.

If the list of recently-deleted paths is empty, the infinite loop blocks indefinitely while waiting for a new event from the watcher. If the list is not empty, the infinite loop will wait for the new event but timeout on the next earliest expiry time in the list. If a timeout occurs, it removes the associated path from the list and waits for the next loop cycle.

5.4.2.2. Implementation

When the handler receives an event, the handler must determine whether the event belongs to a pair of create-delete events. The issue is that the pairing event may appear in a later loop, or it might have already arrived in an earlier loop. As such, the handler stores any received delete events in a list which is initialised on the first loop and persistent across future loops. The handler will search this list for a matching pair on every received create event.

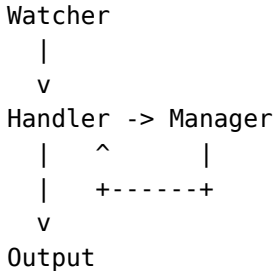
However, we do not want to store these events in the list indefinitely - if the handler receives a remove event for `foo.txt`, then a create event for `foo.txt` 10 hours later, these events should not be paired together as a single rename event. The handler thus adds a timeout to each delete event added to the list. The timeout is an arbitrary short time window, the implementation uses a time window of 10ms. In every loop, the handler will now either wait for a new event from the handler, or wait for the timeout of any path in the list, whichever occurs first. If a timeout occurs, the handler removes the path from the list and waits for the next loop cycle.

In the actual implementation, the timeout of the handler must be handled asynchronously to prevent blocking of the current thread, otherwise new events from the watcher cannot be processed as the handler waits for a matching pair. The implementation overcomes this by starting the watcher and handler in separate threads.

The full code for the event handler is included as an appendix at Section 8.2.

5.4.3. Previous Implementation

The solution had gone through several iterations before reaching the current implementation. I will briefly cover one of the previous iterations.



Listing 2: Diagram showing a previous iteration of the watcher

Compared the the final iteration, this iteration is more complex. The assumption used in this iteration differs from that in the final iteration: A pair of create and delete events within a short time window with the same file name is caused by a file rename event. The assumption does not specify the order of create and delete events, meaning a create followed by a delete event is treated the same as a delete followed by a create event.

The handler in this iteration differs from the final solution in that it only checks if an event can be used as-is, otherwise it sends the event to the manager. The handler is not responsible for pairing create and delete events, but rather the manager.

The manager is an asynchronous infinite loop like the handler, it loops through create and delete events received from the handler. The key difference from the final solution is that this solution adds an expiry time to both delete events and create events. The reason is that this solution assumes that the create and delete events corresponding to a single rename event can be in any order, either create-delete or delete-create. It searches for a pair of matching events upon receipt of either create or delete events.

An error was discovered in the assumption when I was implementing unit tests for the module.

In one of the unit tests, the test first creates several files called `apple.txt` in several subfolders, then removes a random `apple.txt` file. This should not be considered as a rename event. However, the file watcher observes that the deleted `apple.txt` file has the same name as the previously-created `apple.txt` files, and considers the delete event as a rename event.

The assumption fails to consider that in a rename event, a delete event must precede a create event. This means the create / delete events corresponding to a rename event must be ordered. The assumption was corrected and this led to the final iteration described previously.

The code for this iteration is included in the appendix at Section 8.3.

5.5. Software Manual

As part of the evaluation for the project, I have created a public website containing documentation for the software (Wong 2023). The website is hosted on GitHub Pages and contains information such as the software download link, version changelogs, documentation about basic usage, and documentation about the custom query language used by the software.

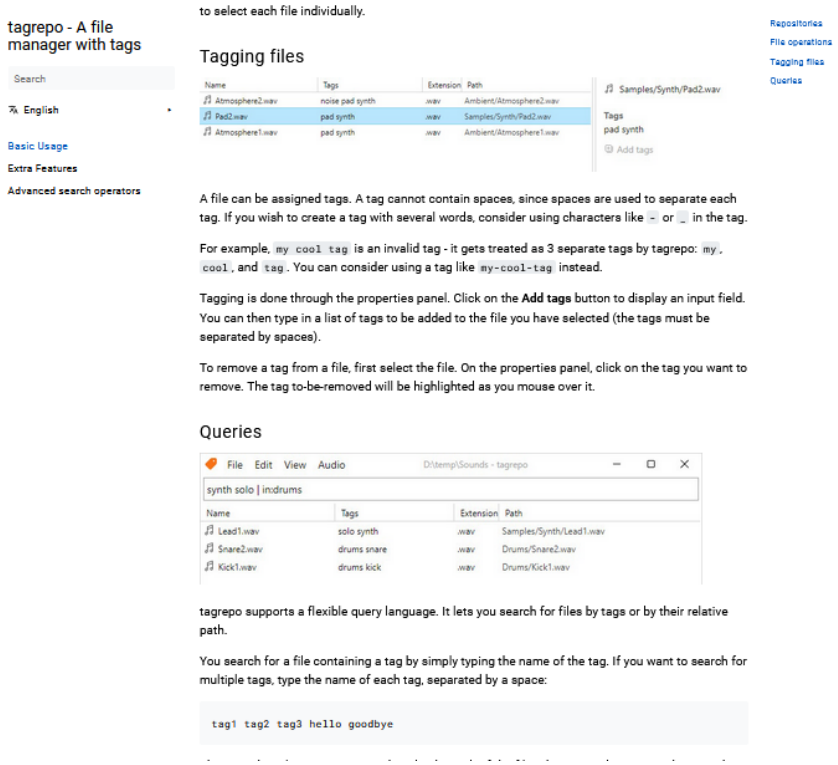


Figure 5: Screenshot of the documentation website, 17 May 2023

One of the requirements of the project was that the software should remain accessible to new users. The manual helps with this requirement by providing explanations of related concepts and software features, allowing new users to gain a better understanding of the software and its uses.

Chapter 6: Evaluation

6.1. Software Testing

Testing of the application involved implementing unit tests for each application module, as well as integration tests for testing how the different modules integrate together. As of writing, 130 unit and integration tests have been implemented. A total of 9129 lines of code were written in this project. The test output is included in the appendix at Section 8.5.

Unit tests were implemented for each module in the Rust backend. The tested modules include `diff.rs` for diffing file trees, `repo.rs` for the repository definition, `scan.rs` for scanning a folder, `sql.rs` for escaping strings for use in SQL queries, and `parser.rs` and `convert.rs` for the query transpiler.

Integration tests were implemented in `query_repo.rs` with a mock repository to assert that the results of the query transpiler were as expected when used in an actual SQLite connection.

6.2. User Testing

User testing was performed in two forms. The first form involved giving a group of participants the software and a task, and finally completing a survey. The second form involved a public release of the software to gather feedback and suggestions about the software. Since this was an evaluation of a service with the aim of improving the service, this project is exempt from needing research ethics approval.

6.2.1. User Trials

A group of five users were given the software and tasked to tag files in a given folder. A brief introduction of the software was given, and a link to the manual described in Section 5.5 was provided. The survey form used in the trials is included as an appendix at Section 8.6. I will discuss the results of the survey in this section.

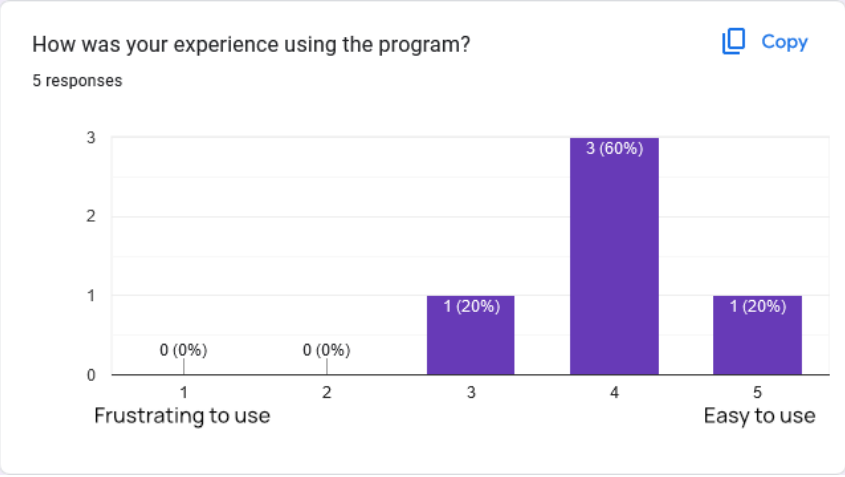


Figure 6: Responses to the question “How was your experience using the program?”

The participants generally felt the program was easy to use. This indicates that the user interface is generally user-friendly and understandable, even when the users had not seen the software before.

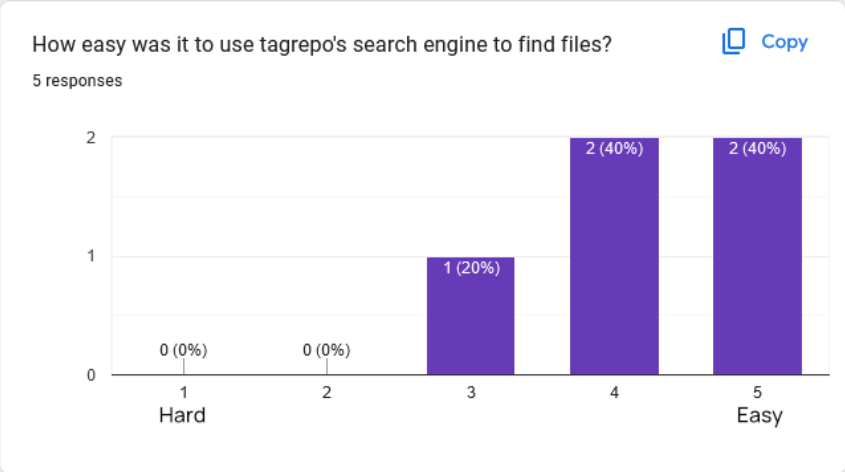


Figure 7: Responses to the question "How easy was it to use tagrepo's search engine to find files?"

Most of the users felt it was easy to use the search engine to search for files.

One user noted that they had expected the search bar to be used to search by file name, not by tag. When a new repository is created and the user has not assigned any tags yet, tag searches on the query bar will yield no results. In order to search for files, one must use prefixed operators such as `in:my_folder` to search for files without using tags. This design decision may be beneficial if the user rarely searches by file name, for example in repositories that are well-tagged. However, this design can become inconvenient in less well-tagged repositories as users rely more on searching by file names, and must prefix each search term with the text `in:`. Solving this issue will require modifications to the custom query language used in the app. One possible solution would be to treat unary terms as filenames instead of tags, for example the query `hello` would search for file paths containing the string "hello" instead of files with the tag "hello". Tag searches would be performed using a prefixed operator like `tag:`. However, this simply reverses the problem as searching becomes inconvenient in well-tagged repositories. This solution could be included as a toggleable option, where users can switch between "tag search mode" and "path search mode". This would require changes to the UI to make it clear which query mode is currently active.

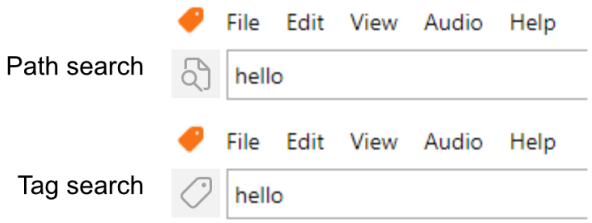


Figure 8: Mock-up UI for toggle between tag search and path search

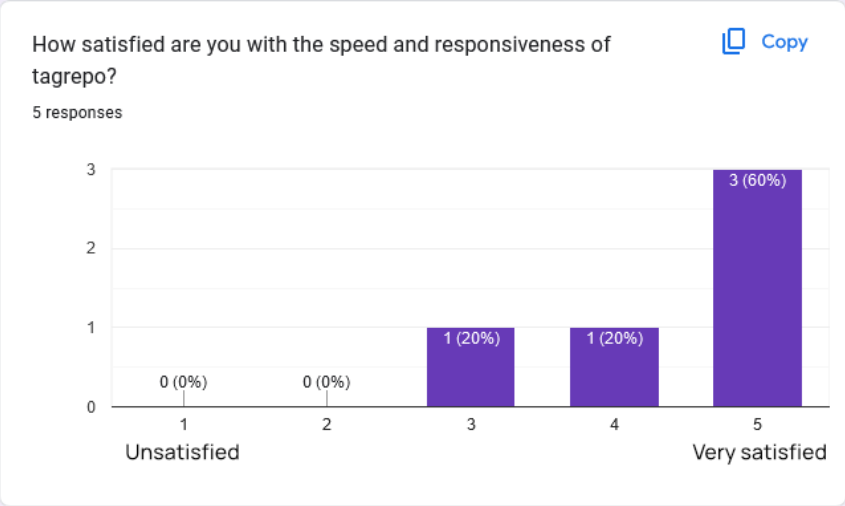


Figure 9: Responses to the question "How satisfied are you with the speed and responsiveness of tagrepo?"

Most of the users felt the software was quick and responsive. This is helped by the implementation of the virtual list and reduced data transfer described in Section 5.3.

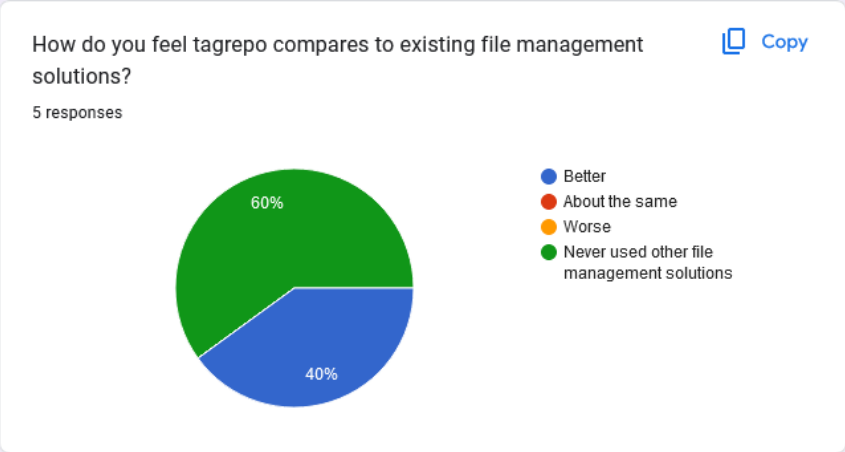


Figure 10: Responses to the question "How do you feel tagrepo compares to existing file management solutions?"

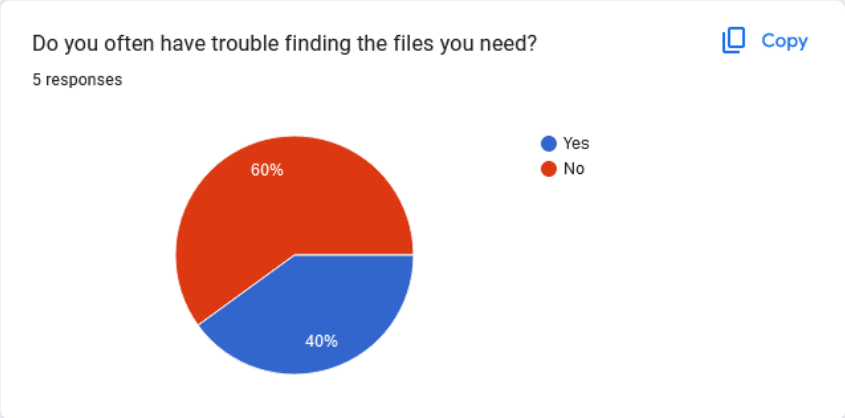


Figure 11: Responses to the question "Do you often have trouble finding the files you need?"

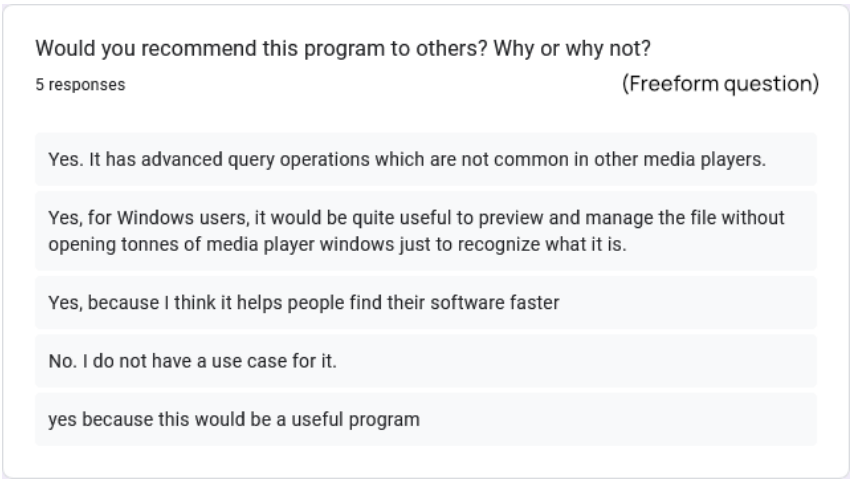


Figure 12: Responses to the free-form question "Would you recommend this program to others? Why or why not?"

While the responses to the questions are generally positive, one of the responses highlight the fact that this software solves the issue of file organisation in a specific way that may not be applicable to all users. However, for users who are dealing with file management issues, the software can provide an efficient solution.

6.2.2. Public Feedback

The software was posted on an online forum called "Reddit", where I received feedback and suggestions from forum users. The forum was divided into many sub-communities, I posted the software in the "datacurator" community, which focuses on file management and organisation. The software was well-received, most of the responses I received were suggestions for new features and use cases. The post had 34 points with 95% upvoted, as well as 21 comments.

One common suggestion was to introduce hierarchies of saved searches, in that they are organised similar to folders. The difference from folders is that a single file can appear in multiple hierarchies at the same time, and these hierarchies may be dynamically populated. One example of this may be having two different hierarchies for a personal photo collection: one hierarchy would be organised according to the persons that appear in the photo, and another one would be organised by the year the photo was taken.

> *I see this as a tool for not only searching data, but also for organizing and presenting it. Create multiple simultaneous alternative hierarchies for the same data. And do so both dynamically or statically. As tags are added, or new hierarchies the same data may appear in different hierarchies at the same time. Different views of the structure deduced from the tagged data.*

Another suggestion was to enable real-time synchronisation between systems. This would allow the same tags to be accessed from multiple systems at the same time, for example on a desktop computer and on a mobile device. The current design of the repository relies on a SQLite database which is a binary data format. If any synchronisation errors occur on multiple devices, it is impossible to resolve conflicts in a binary database compared to using a plain-text tag storage method. One possible solution would be to support multiple tagging methods in the software, one being the currently-used SQLite database method,

and another being a plain-text method that creates a sidecar file next to the file being tagged.

Many of the suggested features were out of scope for this software, such as image deduplication and tag detection from videos. These use cases could be supported with the implementation of a plugin system that allows users to write their own scripts and integration with external programs. This will be further discussed in Section 7.1.2.

The feedback reflects that the design of the software has potential and can be further developed in many ways.

Chapter 7: Conclusion

The aim of this project was to develop a file manager that allowed assigning tags to files while still retaining the existing folder structure. At the same time, the file manager must fulfill a number of requirements listed in Section 3.

The development of the project involved the development of a custom query language that transpiles to an SQL statement, an asynchronous file watcher to handle Windows-specific behaviour in file events, a database schema for efficiently storing and retrieving tags using SQLite, and a frontend with a lazily loaded list that improves performance.

Many challenges arose during the development of the project. One was learning the Rust language from scratch when I had little to no experience of typed languages or low-level languages, concepts such as lifetimes in Rust were particularly difficult to understand since I only had prior experience in scripting languages such as Python. Another major challenge was the implementation of the asynchronous file watcher, I was able to use my knowledge of tasks as loops from embedded systems to implement it successfully, but the process of developing the solution was still very difficult.

In the end, the development of the software was successful. All of the requirements stated previously were met. The feedback received from user testing was well-received, and from that I had obtained new ideas and possible ways to further develop and extend the software.

Overall, the project was a success. I was able to apply my knowledge of operating systems, compilers, databases, web development, and user interfaces in a practical application that can not only solve my own file management problems, but can also help other users in file management. This is a major achievement for me as I was able to finish developing fully-functional software and distribute it to end-users.

7.1. Future Work

The software described in this report is still in its infancy. It is still unfinished and should be further developed to refine existing features, to implement additional functionality, as well as to fix bugs. The feedback obtained from the evaluation is valuable for determining areas of improvement for the software, for example the custom query language and the user interface. I also have ideas for the software that I would like to implement. All these factors have helped refine the plans for further developing the software.

Below are a few examples of future work.

7.1.1. Query Simplification

This extension relates to simplifying the generated SQL statement produced by the parser described in Section 5.1.

Queries expressed in the query bar can be arbitrarily complicated due to the nature of user-submitted inputs. The software currently translates the plain-text query into an SQL statement without any optimisation. In SQLite, FTS terms are the most expensive to compute because of the usage of subqueries as mentioned in Section 5.1.1.3. Therefore, the number of FTS terms in the resulting SQL query should be minimised.

The following example query contains tags (`a`, `b`, `c`) and path (`in:1`, `in:2`) terms, the output SQL query contains three separate FTS terms in total.

```
-- The plain-text query
(a in:1 | b in:1) c

-- The generated SQL WHERE clause
WHERE (
  (
    (i.id IN (SELECT id FROM tag_query('a'))
      AND i.path LIKE '1%')
    OR
    (i.id IN (SELECT id FROM tag_query('b'))
      AND i.path LIKE '1%')
  )
  AND
  i.id IN (SELECT id FROM tag_query('C'))
)
```

To optimise the query, we can convert the input query into conjunctive normal form (CNF) and disjunctive normal form (DNF) (Büning and Lettmann 1999). These forms flatten any nested groupings in the query, and represent the query as either an AND of ORs, or an OR of ANDs. Depending on the query submitted by the user, either the CNF or DNF form will contain the least amount of query terms.

For instance, the query expressed above can be converted into DNF, which yields the following SQL statement with only two FTS terms:

```
-- The plain-text query in DNF
a c in:1 | b c in:1

-- The generated SQL WHERE clause
WHERE (
  (i.id IN (SELECT id FROM tag_query('a AND c'))
    AND i.path LIKE '1%')
  OR
  (i.id IN (SELECT id FROM tag_query('b AND c'))
    AND i.path LIKE '1%')
)
```

Further work on the software will include implementation of CNF and DNF algorithms to simplify the user-provided query before converting to an SQL statement.

7.1.2. Plugin System

As seen in the discussion on public feedback at Section 6.2.2, the software could potentially be used for many diverse use cases. However, it is infeasible to implement features for all possible use cases in a single program.

One approach would be to implement a plugin system for the software, such that users are able to create their own scripts for their own particular use case. This is a popular approach taken by many existing software such as Blender, Visual Studio Code, Google Chrome and many more.

The plugin system should allow editing of file tags stored in the repository, as well as calling external programs to provide integration with other software. This would enable many of the diverse use cases discussed in Section 6.2.2.

There are many choices for choosing a programming language as the scripting interface for the software. One popular choice is Lua, which is used as the plugin system for Neovim. Lua is a fast and lightweight language (Pall 2012) that has good integration with many languages including C and Rust, this has made it a good choice for a plugin system language.

7.1.3. Audio Sample Categorisation

Audio sample categorisation is a task where given an audio file, the goal is to classify it into one or more categories, such as genre, instrument, and mood. This task can be used in various applications, such as automated music recommendation and music search.

In the context of this project, audio sample categorisation can be used to automatically tag and categorise audio samples. This feature can be found in many popular audio sample management software, such as Algonaut Atlas 2 (Sherbourne 2022) and Waves Cosmos (Rogerson 2022).

The audio sample categorisation task is challenging due to the wide variety of possible audio files that can be encountered. This means that the model must be able to handle a wide range of audio files with varying lengths and characteristics. Some common approaches for audio sample categorisation include deep learning techniques such as convolutional and recurrent neural networks.

To implement this extension, machine learning techniques such as convolutional or recurrent neural network models can be used. Obtaining a dataset of annotated audio samples is also one challenge with this extension.

Chapter 8: Appendix

8.1. Database Schema

```
CREATE TABLE items (
  id INTEGER PRIMARY KEY,
  path TEXT UNIQUE NOT NULL,
  tags TEXT NOT NULL,
  meta_tags TEXT NOT NULL DEFAULT 'all'
);

-- an expression index
CREATE INDEX items_path_dirname ON items(dirname(path));
CREATE INDEX items_path_extname ON items(extname(path));

-- FTS5 Documentation:
-- https://www.sqlite.org/fts5.html
CREATE VIRTUAL TABLE tag_query USING fts5 (
  -- Include columns to be stored on this virtual table:
  -- Include the `id` column so I can join it to `items`, but don't index with
FTS
  id UNINDEXED,
  -- Include the `tags` column to index them
  tags,
  -- a 'meta' column that stores additional tags, e.g. 'all'
  meta_tags,

  -- Make this an external content table (don't store the data in this table, but
reference
  -- the original table)
  content=items,
  content_rowid=id,

  -- Use the ascii tokenizer, we want to preserve the original tags as much as
possible
  -- TODO: Implement your own tokenizer that only splits by a single character,
e.g. \x01
  -- https://www.sqlite.org/fts5.html#unicode61_tokenizer
  tokenize="ascii"
);

CREATE TRIGGER items_trigger_ai AFTER INSERT ON items BEGIN
  INSERT INTO tag_query(rowid, tags, meta_tags) VALUES (NEW.id, NEW.tags,
NEW.meta_tags);
END;

CREATE TRIGGER items_trigger_ad AFTER DELETE ON items BEGIN
  INSERT INTO tag_query(tag_query, rowid, tags, meta_tags) VALUES('delete',
OLD.id, OLD.tags, OLD.meta_tags);
END;

CREATE TRIGGER items_trigger_au AFTER UPDATE ON items BEGIN
  INSERT INTO tag_query(tag_query, rowid, tags, meta_tags) VALUES('delete',
OLD.id, OLD.tags, OLD.meta_tags);
```

```
  INSERT INTO tag_query(rowid, tags, meta_tags) VALUES (NEW.id, NEW.tags,
NEW.meta_tags);
END;
```

8.2. Final Iteration of Directory Watcher

```
use std::path::{Path, PathBuf};
use std::time::Duration;

use notify::event::ModifyKind::Name;
use notify::event::{CreateKind, EventAttributes, RemoveKind, RenameMode};
use notify::EventKind::{Create, Modify, Remove};
use notify::{
    Config, Event, EventHandler, ReadDirectoryChangesWatcher, RecursiveMode,
Watcher, WatcherKind,
};
use tokio::sync::mpsc::{unbounded_channel, UnboundedReceiver};
use tokio::time::{timeout_at, Instant};

/// A wrapper for `ReadDirectoryChangesWatcher`.
///
/// The structure of this wrapper is like this:
///
/// ```plain
/// Watcher
///   |
///   v
/// Handler
///   |
///   v
/// Output
/// ```
///
/// The **watcher** is a `Watcher` from the `notify` crate. It spawns events to
be processed by the
/// **handler**.
///
/// The **handler** processes incoming events from the watcher. What it does
depends on the kind of
/// event it received:
///
/// - Delete events: It defers these events for later processing. This is because
both file
///   deletions and file moves are returned as deletions on Windows. This watcher
delays any delete
///   events to see if any create events with the same name are created later,
then treats the event
///   as a rename if so.
/// - Create events: It either returns a create event or a rename event. See the
above point.
/// - Other events: It returns the events as-is.
///
/// ## How to stop watching
///
/// Just drop this struct. It should automatically clean up everything. When this
struct drops, it
/// drops the `notify` watcher, which in turn makes the event handler task stop
since the task
/// is receiving events from the watcher.
```

```
#[derive(Debug)]
pub struct WindowsNormWatcher {
    /// The actual watcher instance.
    watcher: ReadDirectoryChangesWatcher,
}

impl Watcher for WindowsNormWatcher {
    fn new<F: EventHandler>(event_handler: F, config: Config) ->
notify::Result<Self>
    where
        Self: Sized,
    {
        // Spawn the watcher
        let (watcher_tx, watcher_rx) = unbounded_channel();

        let watcher =
            ReadDirectoryChangesWatcher::new(move |res|
watcher_tx.send(res).unwrap(), config)?;

        // Spawn the event handler
        // Don't need to store the JoinHandle, it should naturally terminate once
the watcher drops
        tokio::spawn(async move {
            event_handler_loop(watcher_rx, event_handler).await;
        });

        Ok(Self { watcher })
    }

    fn watch(&mut self, path: &Path, recursive_mode: RecursiveMode) ->
notify::Result<()> {
        self.watcher.watch(path, recursive_mode)
    }

    fn unwatch(&mut self, path: &Path) -> notify::Result<()> {
        self.watcher.unwatch(path)
    }

    fn kind() -> WatcherKind
    where
        Self: Sized,
    {
        WatcherKind::ReadDirectoryChangesWatcher
    }
}

async fn event_handler_loop(
    mut watcher_rx: UnboundedReceiver<notify::Result<Event>>,
    mut event_handler: impl EventHandler,
) {
    fn clear_expired_records(
        recent_deleted_paths: &mut Vec<(Instant, PathBuf, EventAttributes)>,
        event_handler: &mut impl EventHandler,
    ) {
```

```
        let now = Instant::now();
        let mut i = 0;
        loop {
            if i == recent_deleted_paths.len() {
                break;
            }
            {
                let (expires_at, _, _) = recent_deleted_paths.get(i).unwrap();
                if expires_at <= &now {
                    let (_, path, attrs) = recent_deleted_paths.remove(i);
                    let evt = Event {
                        kind: Remove(RemoveKind::Any),
                        paths: vec![path],
                        attrs,
                    };
                    event_handler.handle_event(Ok(evt));
                } else {
                    // not expired, move on to next record
                    i += 1;
                }
            }
        }
    }
    let mut last_rename_from: Option<PathBuf> = None;
    let mut recent_deleted_paths: Vec<(Instant, PathBuf, EventAttributes)> = vec!
[];
    let mut res;
    loop {
        // If we have paths in the database, timeout until the next path's
instant
        if recent_deleted_paths.len() > 0 {
            let next_wake_time = recent_deleted_paths.get(0).unwrap().0;
            match timeout_at(next_wake_time, watcher_rx.recv()).await {
                Ok(x) => {
                    // Didn't timeout, assign the return value to res
                    res = x;
                }
                Err(_) => {
                    // Timeout occurred, clear expired records from database and
wait again
                    clear_expired_records(&mut recent_deleted_paths, &mut
event_handler);
                    continue;
                }
            }
        } else {
            // No paths in database, just wait for next record indefinitely
            res = watcher_rx.recv().await;
        }
        match res {
            Some(evt) => {
                if evt.is_err() {
                    event_handler.handle_event(evt);
                    continue;
```

```
                }
                let evt = evt.unwrap();
                match evt {
                    Event {
                        kind: Modify(Name(RenameMode::From)), mut paths, ..
                    } => {
                        if let Some(_) = last_rename_from {
                            panic!("Got multiple 'Rename From' events in a row!")
                        }
                        let path = paths.pop().unwrap();
                        last_rename_from = Some(path);
                        continue;
                    }
                    Event {
                        kind: Modify(Name(RenameMode::To)),
                        mut paths,
                        attrs,
                    } => {
                        let from_path = last_rename_from.take().expect(
                        "Got 'Rename To' event, but no 'Rename From' event
happened before this!",
                        );
                        let to_path = paths.pop().unwrap();
                        let evt = Event {
                            kind: Modify(Name(RenameMode::Both)),
                            paths: vec![from_path, to_path],
                            attrs,
                        };
                        event_handler.handle_event(Ok(evt));
                    }
                    Event { kind: Remove(RemoveKind::Any), mut paths, attrs } =>
{
                        assert_eq!(
                            paths.len(),
                            1,
                            "Number of created paths is not 1: {}",
                            paths.len()
                        );
                        let removed_path = paths.pop().unwrap();
                        let expires_at = Instant::now() +
Duration::from_millis(10);
                        recent_deleted_paths.push((expires_at, removed_path,
attrs));
                    }
                    Event { kind: Create(CreateKind::Any), mut paths, attrs } =>
{
                        assert_eq!(
                            paths.len(),
                            1,
                            "Number of created paths is not 1: {}",
                            paths.len()
                        );
                        let created_path = paths.pop().unwrap();
                        let mut deleted_path_match_id: Option<usize> = None;
```

```
                        for i in 0..recent_deleted_paths.len() {
                            let deleted_path =
&recent_deleted_paths.get(i).unwrap().1;
                            let created_name = created_path
                                .file_name()
                                .expect("Path doesn't have file name");
                            let deleted_name = deleted_path
                                .file_name()
                                .expect("Path doesn't have file name");
                            if created_name == deleted_name {
                                deleted_path_match_id = Some(i);
                                break;
                            }
                        }
                        match deleted_path_match_id {
                            Some(i) => {
                                let deleted_path_match =
recent_deleted_paths.remove(i).1;
                                let evt = Event {
                                    kind: Modify(Name(RenameMode::Both)),
                                    paths: vec![deleted_path_match,
created_path],
                                    attrs,
                                };
                                event_handler.handle_event(Ok(evt));
                            }
                            None => {
                                let evt = Event {
                                    kind: Create(CreateKind::Any),
                                    paths: vec![created_path],
                                    attrs,
                                };
                                event_handler.handle_event(Ok(evt));
                            }
                        }
                    }
                    _ => event_handler.handle_event(Ok(evt)),
                }
            }
            None => {
                // send remaining deleted paths to output
                for (_, path, attrs) in recent_deleted_paths {
                    let evt = Event {
                        kind: Remove(RemoveKind::Any),
                        paths: vec![path],
                        attrs,
                    };
                    event_handler.handle_event(Ok(evt));
                }
                break;
            }
        }
    }
}
```

8.3. First Iteration of Directory Watcher

```
use std::path::{Path, PathBuf};
use std::time::Duration;

use async_trait::async_trait;
use futures::channel::oneshot;
use notify::event::ModifyKind::Name;
use notify::event::{CreateKind, ModifyKind, RemoveKind, RenameMode};
use notify::EventKind::{Create, Modify, Remove};
use notify::{Config, Event, ReadDirectoryChangesWatcher, RecursiveMode, Watcher};
use tokio::sync::mpsc::{unbounded_channel, UnboundedReceiver, UnboundedSender};
use tokio::task;
use tokio::task::JoinHandle;
use tokio::time::{timeout_at, Instant};

use crate::watch::NormWatcher;

#[derive(Debug, Eq, PartialEq)]
enum PathRecordAction {
    Created,
    Removed,
}

#[derive(Debug)]
enum PathRecordCreationError {
    InvalidPath,
    // InvalidEvent,
}

#[derive(Debug)]
struct PathRecord {
    /// Any path that gets stored here MUST allow calling #file_name without
error
    path: PathBuf,
    action: PathRecordAction,
    sender: oneshot::Sender<ManagerResponse>,
    expires_at: Instant,
}

impl PathRecord {
    fn create(
        path: PathBuf,
        action: PathRecordAction,
        sender: oneshot::Sender<ManagerResponse>,
    ) -> Result<Self, PathRecordCreationError> {
        // verify that path is valid
        path.file_name()
            .ok_or(PathRecordCreationError::InvalidPath)?;

        let expires_at = Instant::now() + Duration::from_millis(100);

        Ok(Self { path, action, sender, expires_at })
    }
}
```

```
#[derive(Debug)]
enum ManagerResponse {
    /// Respond that "This event is not a rename, treat it as the original
create/remove event.
    NotRename,
    /// Respond that "This event is a rename, and create a new rename event".
    CreateRename(PathBuf),
    /// Respond that "This event is a rename, but skip this event", implying the
pairing event will
    /// handle this.
    IgnoreRename,
}

async fn path_records_manager<'a>(mut rx: UnboundedReceiver<PathRecord>) {
    use ManagerResponse::*;

    fn clear_expired_records(db: &mut Vec<PathRecord>) {
        let now = Instant::now();
        let mut i = 0;
        loop {
            if i == db.len() {
                break;
            }
            let mut is_expired = false;
            {
                let x = db.get(i).unwrap();
                if x.expires_at <= now {
                    // record has expired, prepare to remove this record
                    // DON'T increment i to next record
                    is_expired = true;
                } else {
                    // not expired, move on to next record
                    i += 1;
                }
            }
            if is_expired {
                let x = db.remove(i);
                // send event, don't care if the receiver has been dropped
                let _ = x.sender.send(NotRename);
            }
        }
    }

    // TODO: Replace list with a binary heap
    let mut db: Vec<PathRecord> = vec![];
    let mut res;

    loop {
        // If we have paths in the database, timeout until the next path's
instant
        if db.len() > 0 {
            let next_wake_time = db.get(0).unwrap().expires_at;
            match timeout_at(next_wake_time, rx.recv()).await {
```

```
                Ok(x) => {
                    // Didn't timeout, assign the return value to res
                    res = x;
                }
                Err(_) => {
                    // Timeout occurred, clear expired records from database and
wait again
                    clear_expired_records(&mut db);
                    continue;
                }
            }
        } else {
            // No paths in database, just wait for next record indefinitely
            res = rx.recv().await;
        }

        match res {
            Some(record) => {
                // Got instructions to match this path record

                // Scan records to find match
                let mut idx_to_remove = None;
                for (i, other_record) in db.iter().enumerate() {
                    // If both have the same path, and one is Created and other
is Removed...
                    let name_a = record.path.file_name().expect("Path has no
filename");
                    let name_b = other_record.path.file_name().expect("Path has
no filename");
                    if name_a == name_b && record.action != other_record.action {
                        idx_to_remove = Some(i);
                        break;
                    }
                }
                if let Some(i) = idx_to_remove {
                    // Found match, send responses and remove from database
                    let other_record = db.remove(i);
                    record.sender.send(CreateRename(other_record.path)).unwrap();
                    other_record.sender.send(IgnoreRename).unwrap();
                } else {
                    // No match, add to database
                    db.push(record);
                }

                // Clear expired records from database
                clear_expired_records(&mut db);
            }
            None => {
                // No more instructions, all senders have been dropped
                break;
            }
        }
    }
}
```

```
/// A wrapper for `ReadDirectoryChangesWatcher`.
///
/// The structure of this wrapper is like this:
///
/// ```plain
/// Watcher
///   |
///   v
/// Handler -> Manager
///   |   ^      |
///   |   +------+
///   v
/// Output
/// ```
///
/// The **watcher** is a `Watcher` from the `notify` crate. It spawns events to
be processed by the
/// **handler**.
///
/// The **handler** processes incoming events from the watcher. What it does
depends on the kind of
/// event it received:
///
/// - Create/Delete events: It sends these events to the **manager** to be
further processed. It
///   later receives back a message from the **manager** then returns an event
according to the
///   message.
///
///   This is because Windows shows file move events as Create/Delete events. The
**manager** will
///   attempt to resolve these Create/Delete events into Rename events if
possible, otherwise the
///   manager returns the original Create/Delete events.
///
/// - Other events: It returns the events as-is, without sending them to the
manager
///
/// The **manager** maintains a list of recently-created/deleted paths. When a
new path is
/// created/deleted, it scans this list to check if there are any similar
deleted/created path. If
/// so, it tells the **handler** to treat the event as a rename event. If not
found, it adds the
/// path to the list, then tells the **handler** to return the original event as-
is.
///
/// ## How to stop watching
///
/// Just drop this struct. It should automatically clean up everything. If you
really need to drop
/// it manually, here are some instructions / information:
///
```

```rust
/// - Drop the `notify` watcher first. Since the other tasks depend on receiving events from the
///   `notify` watcher, terminating the watcher will naturally cause the tasks to terminate
/// - Then `await` on the two task handlers. You want to ensure that the tasks have really
///   ended.
pub struct ReadDirectoryChangesNormWatcher {
    /// The actual watcher instance.
    watcher: ReadDirectoryChangesWatcher,
    /// Handle for the path record manager / debouncer thing. Its only purpose is to keep the handle
    /// in memory and only drop it when this struct is dropped.
    manager_handle: JoinHandle<()>,
    /// A receiver that receives processed events from the event handler. As the name suggests,
    /// these events are final ("output") events.
    output_rx: UnboundedReceiver<notify::Result<Event>>,
    /// Handle for the event handler. Its only purpose is to keep the handle in memory and only drop
    /// it when this struct is dropped.
    event_handler_handle: JoinHandle<()>,
}

impl ReadDirectoryChangesNormWatcher {
    pub fn new() -> notify::Result<Self> {
        // Spawn the watcher
        let (watcher_tx, watcher_rx) = unbounded_channel();

        let watcher = ReadDirectoryChangesWatcher::new(
            move |res| watcher_tx.send(res).unwrap(),
            Config::default(),
        )?;

        // Spawn the path manager
        let (manager_tx, manager_rx) = unbounded_channel();
        let manager_handle = tokio::spawn(async move {
            path_records_manager(manager_rx).await;
        });

        // Spawn the event handler
        let (output_tx, output_rx) = unbounded_channel();
        let event_handler_handle = tokio::spawn(async move {
            event_handler(watcher_rx, manager_tx, output_tx).await;
        });

        Ok(Self {
            watcher,
            manager_handle,
            output_rx,
            event_handler_handle,
        })
    }
}
```

```
#[async_trait]
impl NormWatcher for ReadDirectoryChangesNormWatcher {
    fn watch(&mut self, path: &Path, recursive_mode: RecursiveMode) ->
notify::Result<()> {
        self.watcher.watch(path.as_ref(), recursive_mode)
    }

    async fn recv(&mut self) -> Option<notify::Result<Event>> {
        self.output_rx.recv().await
    }

    #[cfg(test)]
    fn stop_watching(&mut self) {
        let temp_watcher = ReadDirectoryChangesWatcher::new(|_res| {},
Config::default()).unwrap();
        let real_watcher = std::mem::replace(&mut self.watcher, temp_watcher);
        drop(real_watcher);
    }
}

async fn event_handler(
    mut watcher_rx: UnboundedReceiver<notify::Result<Event>>,
    manager_tx: UnboundedSender<PathRecord>,
    output_tx: UnboundedSender<notify::Result<Event>>,
) {
    let mut last_rename_from: Option<PathBuf> = None;
    while let Some(evt) = watcher_rx.recv().await {
        if evt.is_err() {
            output_tx.send(evt).unwrap();
            continue;
        }
        let evt = evt.unwrap();
        match evt {
            Event {
                kind: Modify(Name(RenameMode::From)), mut paths, ..
            } => {
                if let Some(_) = last_rename_from {
                    panic!("Got multiple 'Rename From' events in a row!")
                }
                let path = paths.pop().unwrap();
                last_rename_from = Some(path);
                continue;
            }
            Event { kind: Modify(Name(RenameMode::To)), mut paths, .. } => {
                let from_path = last_rename_from.take().expect(
                    "Got 'Rename To' event, but no 'Rename From' event happened
before this!",
                );
                let to_path = paths.pop().unwrap();
                let evt = Event {
                    kind: Modify(Name(RenameMode::Both)),
                    paths: vec![from_path, to_path],
                    attrs: evt.attrs.clone(),
```

```
                };
                output_tx.send(Ok(evt)).unwrap();
            }
            Event { kind: Remove(RemoveKind::Any), mut paths, attrs } => {
                assert_eq!(
                    paths.len(),
                    1,
                    "Number of created paths is not 1: {}",
                    paths.len()
                );
                let removed_path = paths.pop().unwrap();
                let (path_tx, path_rx) = oneshot::channel();
                let record =
                    PathRecord::create(removed_path.clone(),
PathRecordAction::Removed, path_tx)
                        .unwrap();
                manager_tx.send(record).unwrap();
                let output_tx = output_tx.clone();

                task::spawn(async move {
                    match path_rx.await {
                        Ok(ManagerResponse::CreateRename(created_path)) => {
                            // found matching create, this is a file-move event
                            // we got a path, meaning we should handle this event
                            // we'll create a rename event:
                            let evt = Event {
                                kind: Modify(Name(RenameMode::Both)),
                                paths: vec![removed_path,
created_path.to_path_buf()],
                                attrs,
                            };
                            output_tx.send(Ok(evt)).unwrap();
                        }
                        Ok(ManagerResponse::IgnoreRename) => {
                            // found matching create, this is a file-move event
                            // however we didn't get a path, meaning the paired
create event will handle this
                            // we'll do nothing here
                        }
                        Ok(ManagerResponse::NotRename) | Err(_) => {
                            // Case (a): no paired path found, treat this as a
remove
                            // Case (b): sender got dropped, the watcher has
likely been dropped,
                            //           just treat it as a normal event
                            let evt = Event {
                                kind: Remove(RemoveKind::Any),
                                paths: vec![removed_path],
                                attrs,
                            };
                            output_tx.send(Ok(evt)).unwrap();
                        }
                    }
                });
```

```
            }
            Event { kind: Create(CreateKind::Any), paths, attrs } => {
                assert_eq!(
                    paths.len(),
                    1,
                    "Number of created paths is not 1: {}",
                    paths.len()
                );
                let created_path = paths.get(0).unwrap().clone();
                let (path_tx, path_rx) = oneshot::channel();
                let record =
                    PathRecord::create(created_path.clone(),
PathRecordAction::Created, path_tx)
                        .unwrap();
                manager_tx.send(record).unwrap();
                let output_tx = output_tx.clone();

                task::spawn(async move {
                    match path_rx.await {
                        Ok(ManagerResponse::CreateRename(removed_path)) => {
                            // found matching remove, this is a file-move event
                            // we got a path, meaning we should handle this event
                            // we'll create a rename event:
                            let evt = Event {
                                kind: Modify(Name(RenameMode::Both)),
                                paths: vec![removed_path, created_path],
                                attrs,
                            };
                            output_tx.send(Ok(evt)).unwrap();
                        }
                        Ok(ManagerResponse::IgnoreRename) => {
                            // found matching remove, this is a file-move event
                            // however we didn't get a path, meaning the paired
remove event will handle this
                            // we'll do nothing here
                        }
                        Ok(ManagerResponse::NotRename) | Err(_) => {
                            // Case (a): no paired path found, treat this as a
create
                            // Case (b): sender got dropped, the watcher has
likely been dropped,
                            //           just treat it as a normal event
                            let evt = Event {
                                kind: Create(CreateKind::Any),
                                paths: vec![created_path.to_path_buf()],
                                attrs,
                            };
                            output_tx.send(Ok(evt)).unwrap();
                        }
                    }
                });
            }
            Event { kind: Modify(ModifyKind::Any), .. } => {
                output_tx.send(Ok(evt)).unwrap();
```

```
                }
                _ => output_tx.send(Ok(evt)).unwrap(),
            }
        }
    }
```

8.4. Virtual List Implementation

```
<script lang="ts" setup>
import ItemListHeader from "@/components/itemlist/ItemListHeader.vue";
import ItemRow from "./itemlist/ItemRow.vue";
import { actions, selection, state } from "@/lib/api";
import { createEventListenerRegistry } from "@/lib/utils";
import { getSpacingSize } from "@/lib/tailwindcss";
import { computed, onBeforeUnmount, onMounted, ref, Ref } from "vue";
import ContextMenu from "@/components/ContextMenu.vue";
import { CopyFilePath, OpenFile, RevealFile } from "@/lib/icons";
import MenuItem from "@/components/menu/MenuItem.vue";
import MenuSeparator from "@/components/menu/MenuSeparator.vue";
import { launchSelectedItems } from "@/lib/api/actions";

const container: Ref<HTMLDivElement | null> = ref(null);

const listeners = createEventListenerRegistry();
let observer: ResizeObserver | null = null;

// width of the viewport
const viewWidth: Ref<number> = ref(0);
// height of the viewport
// NOTE: this includes the header as well.
// to exclude the header, use virtualViewHeight
const viewHeight: Ref<number> = ref(0);

function updateViewSize(container: HTMLDivElement) {
  viewWidth.value = Math.round(container.clientWidth);
  viewHeight.value = Math.round(container.clientHeight);
}

const scrollTop: Ref<number> = ref(0);
const scrollLeft: Ref<number> = ref(0);

function updateScrollPosition(container: HTMLDivElement) {
  scrollTop.value = Math.round(container.scrollTop);
  scrollLeft.value = Math.round(container.scrollLeft);
}

onMounted(() => {
  // component is mounted, container MUST be a div at this point
  const con = container.value!;

  // Detect viewport size of the container
  updateViewSize(con);
  observer = new ResizeObserver((entries) => {
    // the only purpose of this section is to check if the container has been
resized
    let containerWasResized = false;
    for (const entry of entries) {
      if (entry.target !== con) {
        console.warn("got entry for wrong target!", entry.target);
        continue;
      }
```

```
      if (entry.borderBoxSize.length !== 1) {
        console.warn(
          "expected only 1 size, but got not a 1!",
          entry.borderBoxSize.length
        );
        continue;
      }
      containerWasResized = true;
      break;
    }
    // if it's been resized, update the refs
    if (containerWasResized) {
      updateViewSize(con);
    }
  });
  observer.observe(con);

  // Detect scroll position within the container
  updateScrollPosition(con);
  listeners.add(con, "scroll", (evt: Event) => {
    updateScrollPosition(con);
  });
});

onBeforeUnmount(() => {
  // component hasn't been mounted yet, container MUST be a div at this point
  listeners.clear();
  observer?.disconnect();
});

// set item height to Tailwind's 'h-6'
// keep this in sync with ItemRow's height
const itemHeight = getSpacingSize("6");
const headerHeight = getSpacingSize("6");

// the actual amount of vertical space that's rendering items
// you need to subtract the header
const virtualViewHeight = computed(() => viewHeight.value - headerHeight);

const containerHeight = computed(
  () => headerHeight + state.itemIds.length * itemHeight
);
const containerWidth = computed(() =>
  state.listViewColumns.reduce((acc, col) => acc + col.width, 0)
);

const preloadPadding = itemHeight * 10; // px

const indexRangeToRender = computed(() => {
  const renderTop = scrollTop.value - preloadPadding;
  const renderBottom =
    scrollTop.value + virtualViewHeight.value + preloadPadding;
  const itemsBeforeTop = Math.floor(renderTop / itemHeight);
  const itemsUntilBottom = Math.ceil(renderBottom / itemHeight);
```

```
  // fix - if the view is out of bounds (way below the list of items)
  // a bug occurs when you execute a new query while scrolled down, and the new
list is shorter than the previous
  // you need to limit BOTH the start and end index with BOTH max and min values
  const actualItemsCount = state.itemIds.length;
  // subtract 1 here, we're now returning indexes that start from 0, so it's
(item count - 1)
  const startIndex = Math.max(0, Math.min(itemsBeforeTop - 1, actualItemsCount));
  // don't subtract 1 here, because a for-loop ends before the last value
  const endIndex = Math.max(0, Math.min(itemsUntilBottom, actualItemsCount));

  return [startIndex, endIndex];
});

const debug = false;

const menu = ref<InstanceType<typeof ContextMenu> | null>(null);

function scrollToIndex(index: number) {
  const el = container.value;
  if (el === null) return;

  const viewTop = el.scrollTop;
  const viewBottom = el.scrollTop + virtualViewHeight.value;

  const itemTop = index * itemHeight;
  const itemBottom = (index + 1) * itemHeight;

  if (itemTop < viewTop) {
    // item is above the view
    el.scrollTop = itemTop;
  } else if (itemBottom > viewBottom) {
    // item is below the view
    el.scrollTop = itemBottom - virtualViewHeight.value;
  } else {
    // item is in view, do nothing
  }
}

function scrollToFocusedIndex() {
  const focusedIndex = selection.focusedIndex();
  if (focusedIndex !== null) {
    scrollToIndex(focusedIndex);
  }
}

defineExpose({
  focus: () => {
    if (state.itemIds.length > 0) {
      selection.isolate(0);
      scrollToFocusedIndex();
    }
    container.value?.focus();
```

```
  },
});

const log = console.log;
</script>

<template>
  <div
    ref="container"
    class="relative h-full w-full overflow-auto border-r-2 border-white text-sm
focus:outline-none"
    @click="
      (e) => {
        // This is disabled for now, due to a bug.
        // If you drag your mouse across multiple rows, it gets treated as a
click on
        // this element.
        // selection.clear();
      }
    "
    @contextmenu.prevent.stop="(e) => menu?.show(e)"
    tabindex="-1"
    @keydown.enter="launchSelectedItems"
    @keydown.up.prevent="
      (e) => {
        if (e.shiftKey) {
          selection.extendUp();
        } else {
          selection.isolateUp();
        }
        scrollToFocusedIndex();
      }
    "
    @keydown.down.prevent="
      (e) => {
        if (e.shiftKey) {
          selection.extendDown();
        } else {
          selection.isolateDown();
        }
        scrollToFocusedIndex();
      }
    "
    @keydown="
      (e) => {
        // don't do anything if there's nothing in the list
        if (state.itemIds.length === 0) return;

        if (e.key === 'Home') {
          selection.isolate(0);
          scrollToFocusedIndex();
        } else if (e.key === 'End') {
          selection.isolate(state.itemIds.length - 1);
          scrollToFocusedIndex();
```

```
        } else if (e.key === 'a') {
          if (e.ctrlKey) {
            selection.selectAll();
          }
        } else {
          // event is not handled, return early
          return;
        }

        e.preventDefault();
      }
    "
  >
    <!-- The container resizer, it's a 1px div located at the bottom right corner
-->
    <component
      is="div"
      class="absolute -z-10 h-px w-px bg-red-500 opacity-0"
      :style="{
        top: containerHeight - 1 + 'px',
        left: containerWidth - 1 + 'px',
      }"
    />
    <ItemRow
      v-for="n in indexRangeToRender[1] - indexRangeToRender[0]"
      :id="state.itemIds[n + indexRangeToRender[0] - 1]"
      :listIndex="n + indexRangeToRender[0] - 1"
      class="absolute"
      :style="{
        top: `${(n + indexRangeToRender[0] - 1) * itemHeight + headerHeight}px`,
      }"
      :key="state.itemIds[n + indexRangeToRender[0] - 1]"
    />
    <div
      class="fixed bottom-2 right-2 border bg-white opacity-50 drop-shadow"
      v-if="debug"
    >
      {{ indexRangeToRender }}
      <template
        class="mr-1"
        v-for="n in indexRangeToRender[1] - indexRangeToRender[0]"
      >
        {{ n + indexRangeToRender[0] - 1 }}{{ " " }}
      </template>
    </div>
    <!-- I put the header after the items to make it appear above the items -->
    <ItemListHeader />
    <ContextMenu ref="menu" v-slot="{ closeMenu }">
      <MenuItem
        :text="
          selection.selectedCount.value === 1 ? 'Open' : 'Open selected files'
        "
        @click="
          (e) => {
```

```
            actions.launchSelectedItems();
            closeMenu();
          }
        "
      >
        <template #icon="{ defaultClasses }">
          <OpenFile class="h-16px w-16px" :class="defaultClasses" />
        </template>
      </MenuItem>
      <MenuItem
        :text="
          selection.selectedCount.value === 1
            ? 'Reveal in folder'
            : 'Reveal files in folder'
        "
        @click="
          (e) => {
            actions.revealSelectedItems();
            closeMenu();
          }
        "
      >
        <template #icon="{ defaultClasses }">
          <RevealFile class="h-16px w-16px" :class="defaultClasses" />
        </template>
      </MenuItem>
      <MenuSeparator />
      <MenuItem
        :text="selection.selectedCount.value === 1 ? 'Copy path' : 'Copy paths'"
        @click="
          (e) => {
            actions.copySelectedItemPaths();
            closeMenu();
          }
        "
      >
        <template #icon="{ defaultClasses }">
          <CopyFilePath class="h-16px w-16px" :class="defaultClasses" />
        </template>
      </MenuItem>
    </ContextMenu>
  </div>
</template>
```

8.5. Test Output

```
test diff::tests::diff_1 ... ok
test diff::tests::diff_2 ... ok
test diff::tests::diff_3 ... ok
test diff::tests::diff_4 ... ok
test diff::tests::diff_5 ... ok
test diff::tests::diff_6 ... ok
test diff::tests::simpaths_1 ... ok
test diff::tests::simpaths_2 ... ok
test diff::tests::simpaths_3 ... ok
test diff::tests::simpaths_4 ... ok
test helpers::sql::test_fts5::both_quotes ... ok
test helpers::sql::test_fts5::double_quotes ... ok
test helpers::sql::test_fts5::no_quotes ... ok
test helpers::sql::test_fts5::single_quotes ... ok
test helpers::sql::test_like::both_quotes ... ok
test helpers::sql::test_like::double_quotes ... ok
test helpers::sql::test_like::escape_char_1 ... ok
test helpers::sql::test_like::escape_char_2 ... ok
test helpers::sql::test_like::no_quotes ... ok
test helpers::sql::test_like::percent_1 ... ok
test helpers::sql::test_like::percent_2 ... ok
test helpers::sql::test_like::single_quotes ... ok
test helpers::sql::test_like::underscore_1 ... ok
test helpers::sql::test_like::underscore_2 ... ok
test query::convert::test_clauses::common_1 ... ok
test query::convert::test_clauses::common_2 ... ok
test query::convert::test_clauses::common_3 ... ok
test query::convert::test_clauses::fts_1 ... ok
test query::convert::test_clauses::fts_2 ... ok
test query::convert::test_clauses::fts_3 ... ok
test query::convert::test_clauses::fts_4 ... ok
test query::convert::test_clauses::fts_5 ... ok
test query::convert::test_clauses::inpath_1 ... ok
test query::convert::test_clauses::inpath_2 ... ok
test query::convert::test_clauses::inpath_3 ... ok
test query::convert::test_clauses::inpath_4 ... ok
test query::convert::test_clauses::inpath_5 ... ok
test query::convert::test_clauses::inpath_6 ... ok
test query::convert::test_fts_query::and_1 ... ok
test query::convert::test_fts_query::and_2 ... ok
test query::convert::test_fts_query::complex_1 ... ok
test query::convert::test_fts_query::neg_1 ... ok
test query::convert::test_fts_query::neg_2 ... ok
test query::convert::test_fts_query::neg_3 ... ok
test query::convert::test_fts_query::neg_4 ... ok
test query::convert::test_fts_query::neg_5 ... ok
test query::convert::test_fts_query::or_1 ... ok
test query::convert::test_fts_query::or_2 ... ok
test query::convert::test_to_sql::common_1 ... ok
test query::convert::test_to_sql::fts_1 ... ok
test query::convert::test_to_sql::fts_2 ... ok
test query::convert::test_to_sql::fts_3 ... ok
test query::convert::test_to_sql::indir_1 ... ok
```

```
test query::convert::test_to_sql::indir_2 ... ok
test query::convert::test_to_sql::indir_3 ... ok
test query::convert::test_to_sql::indir_4 ... ok
test query::convert::test_to_sql::indir_5 ... ok
test query::parser::expr_tests::and_or_1 ... ok
test query::parser::expr_tests::and_or_2 ... ok
test query::parser::expr_tests::cjk01 ... ok
test query::parser::expr_tests::cjk02 ... ok
test query::parser::expr_tests::common_1 ... ok
test query::parser::expr_tests::complex_1 ... ok
test query::parser::expr_tests::just_and_1 ... ok
test query::parser::expr_tests::just_and_2 ... ok
test query::parser::expr_tests::just_and_3 ... ok
test query::parser::expr_tests::just_and_4 ... ok
test query::parser::expr_tests::just_or_1 ... ok
test query::parser::expr_tests::just_or_2 ... ok
test query::parser::expr_tests::just_or_3 ... ok
test query::parser::expr_tests::just_or_4 ... ok
test query::parser::expr_tests::not_1 ... ok
test query::parser::expr_tests::not_2 ... ok
test query::parser::expr_tests::parens_1 ... ok
test query::parser::expr_tests::parens_2 ... ok
test query::parser::expr_tests::parens_3 ... ok
test query::parser::expr_tests::parens_4 ... ok
test query::parser::expr_tests::string_tags_1 ... ok
test query::parser::expr_tests::string_tags_2 ... ok
test query::parser::expr_tests::string_tags_3 ... ok
test query::parser::expr_tests::unicode ... ok
test query::parser::tests::test_key_value ... ok
test query::parser::tests::test_literal ... ok
test query::parser::tests::test_string ... ok
test query::parser::tests::test_tag ... ok
test query::tests::common_1 ... ok
test query::tests::common_2 ... ok
test query::tests::common_3 ... ok
test query::tests::empty ... ok
test repo::tests::can_batch_insert_tags_1 ... ok
test repo::tests::can_batch_insert_tags_then_query ... ok
test repo::tests::can_batch_remove_tags_1 ... ok
test repo::tests::can_get_all_items ... ok
test repo::tests::can_get_item_by_id ... ok
test repo::tests::can_get_item_by_path ... ok
test repo::tests::can_insert_items ... ok
test repo::tests::can_insert_tags_1 ... ok
test repo::tests::can_query_items ... ok
test repo::tests::can_remove_item_by_id ... ok
test repo::tests::can_remove_item_by_path ... ok
test repo::tests::can_remove_tags_1 ... ok
test repo::tests::can_update_item_path ... ok
test repo::tests::can_update_item_tags ... ok
test repo::tests::cant_insert_duplicate_items ... ok
test repo::tests::check_tables_of_newly_created_database ... ok
test repo::tests::custom_insert_tags_1 ... ok
test repo::tests::custom_insert_tags_2 ... ok
```

```
test repo::tests::custom_remove_tags_1 ... ok
test repo::tests::custom_remove_tags_2 ... ok
test repo::tests::custom_validate_tags_1 ... ok
test repo::tests::custom_validate_tags_2 ... ok
test repo::tests::custom_validate_tags_3 ... ok
test repo::tests::query_test ... ok
test repo::tests::query_test_2 ... ok
test scan::tests::ignores_files_in_folder ... ok
test scan::tests::scans_files_in_folder ... ok
test tests::query_repo::query_1 ... ok
test tests::query_repo::query_2 ... ok
test tests::query_repo::query_3 ... ok
test tests::query_repo::query_4 ... ok
test watch::tests::basic_test ... ok
test watch::tests::file_creations_01 ... ok
test watch::tests::file_creations_02 ... ok
test watch::tests::file_removes_01 ... ok
test watch::tests::file_removes_02 ... ok
test watch::tests::file_renames_01 ... ok
test watch::tests::file_renames_02 ... ok
test watch::tests::file_renames_03 ... ok
test testsa::my_test ... ok
test tree::tests::paths_test ... ok
```

8.6. Evaluation Form

```
= tagrepo Evaluation Form

Thank you for trying out tagrepo!

tagrepo was built as part of my Computer Science course's final project. The
software is intended to provide users an easy-to-use method of organising files
while still maintaining flexibility of use and compatibility with any existing
folder structure.

I am currently gathering feedback for evaluating the project, I would greatly
appreciate it if you could take a moment to fill out this survey.

* Indicates required question

== User experience

1. How was your experience using the program? *
Mark only one oval.
(Frustrating to use) 1 2 3 4 5 (Easy to use)

2. How easy was it to use tagrepo's search engine to find files? *
Mark only one oval.
(Hard) 1 2 3 4 5 (Easy)

3. How satisfied are you with the speed and responsiveness of tagrepo? *
Mark only one oval.
(Unsatisfied) 1 2 3 4 5 (Very satisfied)

4. Would you recommend this program to others? Why or why not? *
(Freeform question)

== User preference

5. How do you feel tagrepo compares to existing file management solutions? *
Mark only one oval.
- [ ] Better
- [ ] About the same
- [ ] Worse
- [ ] Never used other file management solutions

6. Do you often have trouble finding the files you need? *
Mark only one oval.
- [ ] Yes
- [ ] No
- [ ] Other (Freeform answer)

7. Thank you for taking the time to complete this form! If you have any
additional feedback about the software, please feel free to let me know here.
(Freeform question)
```

Bibliography

Balasubramanian, Abhiram et al. 2017. "System Programming in Rust: Beyond Safety." In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 156–161.

Bensing, Dustin. 2019. *#rust2019 Are We GUI Yet?*, accessed November 21, 2022, https://www.areweguiyet.com/.

Bergman, Ofer et al. 2010. "The Effect of Folder Structure on Personal File Navigation." *Journal of the American Society for Information Science and Technology* 61, no. 12. Wiley Online Library: 2426–2441.

Büning, H.K., and T. Lettmann. 1999. *Propositional Logic: Deduction and Algorithms*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, https://books.google.co.uk/books?id=3oJE9yczr3EC.

Cornes, Brad. 2020. *Preflight - Tailwind Css*, accessed April 15, 2023, https://tailwindcss.com/docs/preflight.

Couprie, Geoffroy. 2021. *Nom, Eating Data Byte by Byte*, accessed April 15, 2023, https://github.com/rust-bakery/nom.

Davison, Lawrence. 2022. *Adoption of Rust: Who's Using It and How*, accessed November 21, 2022, https://thenewstack.io/adoption-of-rust-whos-using-it-and-how/.

De Vocht, Laurens et al. 2012. *Formal Experiment Report: Tagging Files Vs. Placing Files in a Hierarchy*. Graz, Austria: Graz University of Technology, https://github.com/novoid/2011-01-tagstore-formal-experiment/blob/master/analysis_and_derived_data/Results_Report.pdf.

Dinneen, Jesse David, and Charles-Antoine Julien. 2020. "The Ubiquitous Digital File: A Review of File Management Research." *Journal of the Association for Information Science and Technology* 71, no. 1. Wiley Online Library: 1–32.

Henderson, Sarah. 2011. "Document Duplication: How Users (Struggle To) Manage File Copies and Versions." *Proceedings of the American Society for Information Science and Technology* 48, no. 1. Wiley Online Library: 1–10.

Henderson, Sarah, and Ananth Srinivasan. 2009. "An Empirical Analysis of Personal Digital Document Structures." In *Symposium on Human Interface*, 394–403.

Hicks, Ben J. et al. 2008. "Organizing and Managing Personal Electronic Files: A Mechanical Engineer's Perspective." *ACM Trans. Inf. Syst.* 26, no. 4, 1–40. https://doi.org/10.1145/1402256.1402262.

Jones, William. 2010. *Keeping Found Things Found: The Study and Practice of Personal Information Management*. Morgan Kaufmann.

Jones, William P, and Jaime Teevan. 2007. *Personal Information Management*. Vol. 14. University of Washington Press Seattle.

Pall, Mike. 2012. *Luajit*, accessed April 18, 2023, https://luajit.org/luajit.html.

Proven, Liam. 2022. *Rust Is Coming to the Linux Kernel*, accessed November 21, 2022, https://www.theregister.com/2022/09/16/rust_in_the_linux_kernel/.

Rogerson, Ben. 2022. *Waves Cosmos Is a Free Sample Management Tool That Organises All Your Loops and One-Shots and Then Lets You Search for the Sound You'Re Looking For*, accessed April 17, 2023, https://www.musicradar.com/news/waves-cosmos-free-sample-management-software.

Sherbourne, Simon. 2022. *Algonaut Atlas 2*, accessed April 17, 2023, https://www.soundonsound.com/reviews/algonaut-atlas-2.

Smith, Gene. 2007. *Tagging: People-Powered Metadata for the Social Web*. New Riders.

Stack Overflow. 2022. *Stack Overflow Developer Survey 2022*, accessed November 28, 2022, https://survey.stackoverflow.co/2022/.

Tauri Contributors. 2022. *Tauri-Apps/tauri*, accessed November 21, 2022, https://github.com/tauri-apps/tauri.

Thain, D. 2019. *Introduction to Compilers and Language Design*. Lulu.com, https://books.google.co.uk/books?id=5mVyDwAAQBAJ.

Wong, Chun Ho. 2023. *Tagrepo*, accessed April 17, 2023, https://jameswalker55.github.io/tag-repo-site/.