An Analysis of Different File Storage Options for use With the ZOIA Library App

Retrieval Of Patches

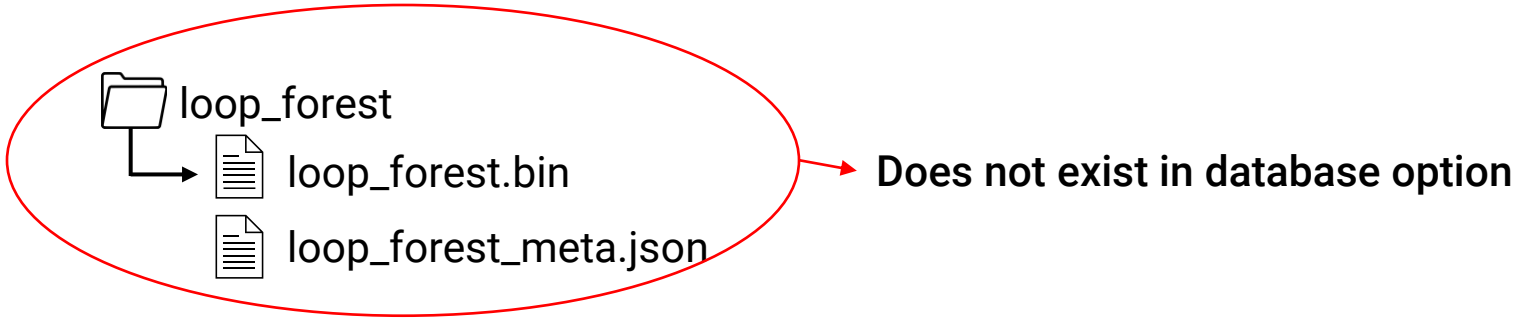
Important Notes

ZOIA Patches on PatchStorage are either stored as .bin or as a compressed file format (.zip, .7z, .tar, .gz, etc.). In either case, the data that can be retrieved from PatchStorage includes the following (ideally everything is preserved except the “platform” since we can drop it after verifying it is for a ZOIA):

Example 1 – Standard .bin Download

The patch 002_zoia_loop_forest.bin is stored on PatchStorage.

- Initially, the Library App will query PatchStorage for necessary patch information. This consists of:
  - The name of the patch
  - The author
  - The date it was modified
  - Any patch notes (if they exist)
  - **The licence**
  - A preview link (if one exists, typically leads to a YouTube video)
  - The state of the patch (Work In Progress, Help Needed, Ready To Go, Inactive)
  - The categories associated with the patch
  - The tags associated with the patch
- If the user chooses to download the patch, only then is the .bin patch file retrieved.
- **Options**:
  - A JSON file is created to store the metadata associated with this patch.
  - The metadata and the associated patch is inserted into a pre-existing database.
- For JSON, upon retrieval of the .bin file, it is stored in a directory that shares a name with the patch. Note that the prefix number and "_zoia_" is dropped:



Example 1 – Standard .bin Download

Here is an example of the contents contained within the loop_forest_meta.json file:


```
{
    "id": 1
    "link": "https://patchstorage.com/loop-forest/",
    "created_at": "2019-08-29",
    "updated_at": "2019-09-17",
    "slug": "loop-forest",
    "title": "Loop Forest",
    "excerpt": "4 separate loopers with various parameters being modulated by LFOs, including record, loop start, reverse…",
    "content": "4 separate loopers with various parameters being modulated by LFOs, including record, loop start, reverse, an
    "files": [
      {
        "id": 1
        "url": "https://patchstorage.com/wp-content/uploads/2019/09/002_zoia_loop_forest.bin"
        "filesize": 32
        "filename": "loop_forest.bin"
      }
    ],
    "preview_url": "https://soundcloud.com/jeremyblake/rmr-zoia-patch-demo-loop"
    "revision": "1",
    "like_count": 28,
    "download_count": 744,
    "author": {
        "id": 1,
        "slug": "red-means-recording",
        "name": "Red Means Recording"
    },
    "categories": [
      {
        "id": 1,
        "slug": "effect",
        "name": "Effect"
      }
    ],
```

Example 1 – Standard .bin Download

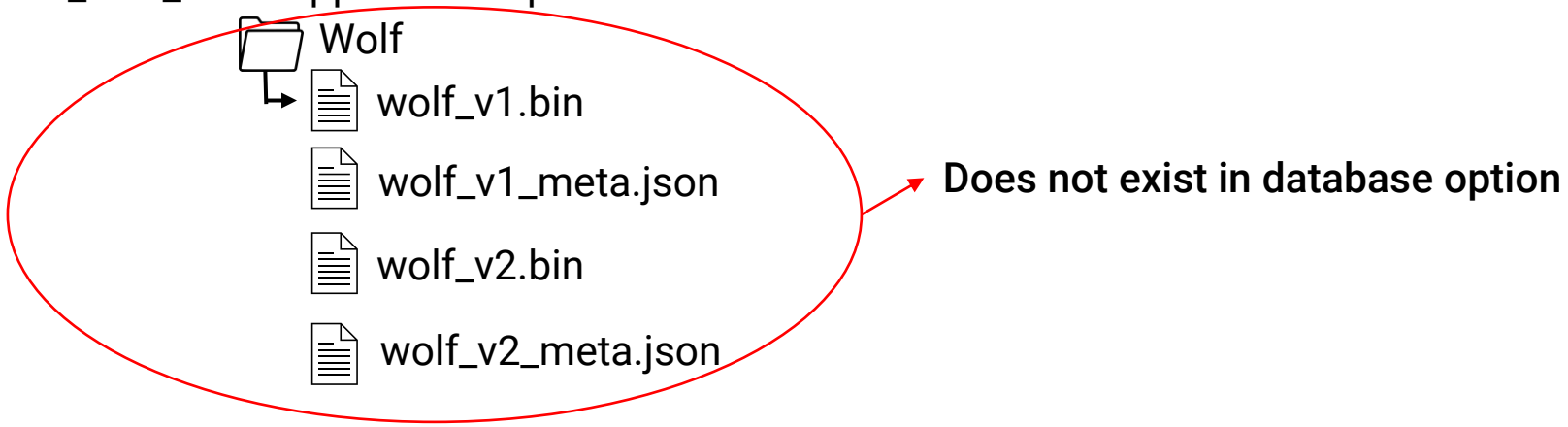
Here is an example of the contents contained within the loop_forest_meta.json file:


```
"categories": [
  {
    "id": 1,
    "slug": "effect",
    "name": "Effect"
  }
],
"tags": [
  {
    "id": 1
    "slug": "ambient"
    "name": "Ambient"
  },
  {
    "id": 2
    "slug": "delay"
    "name": "Delay"
  },
  {
    "id": 3
    "slug": "effect"
    "name": "Effect"
  },
  {
    "id": 4
    "slug": "looper"
    "name": "Looper"
  },
  {
    "id": 5
    "slug": "reverb"
    "name": "Reverb"
  },
],
"state": {
    "id": 1,
    "slug": "ready-to-go",
    "name": "Ready To Go"
},
"license": {
    "id": 1,
    "slug": "gnu-lesser-general-public-license-family",
    "name": "GNU Lesser General Public License family"
},
"custom_license_text": null
}
```

Example 2 – Standard .zip Download

The patch Wolf.zip is stored on PatchStorage.

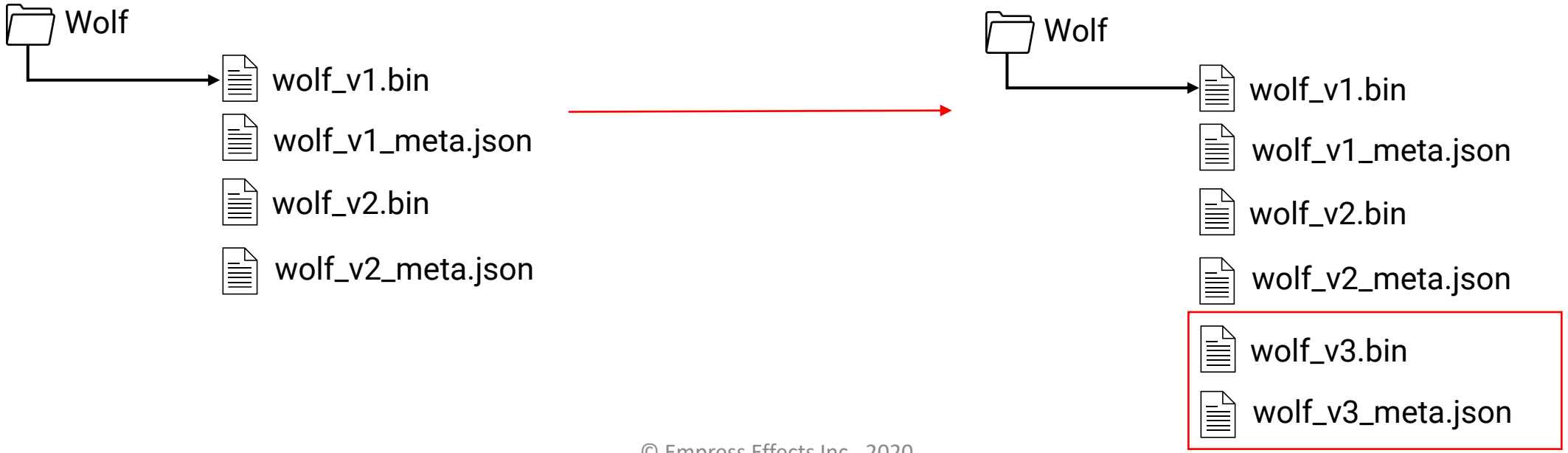
- Initially, the Library App will query PatchStorage for necessary patch information. This mirrors the procedure for a .bin file.
- The .zip is decompressed and each file is analyzed.
- For every .bin file encountered, the following should occur:
  - **Options**:
    - A JSON file is created to store the metadata associated with each patch contained within the .zip.
    - The metadata for each patch contained within the .zip and associated patch itself are inserted into a pre-existing database.
- For every other file encountered:
  - Store the file in a directory that shares the name of the .zip retrieved. The Library App should offer options to interface with common files (.txt, .rtf), but otherwise, it will just list the file, and users can try to open it with a different application.
- For JSON, once all files within the .zip are processed, they are stored in a directory that shares the name with the .zip file. Note that the prefix number and "_zoia_" is dropped for all patches:



Example 3 - Version History

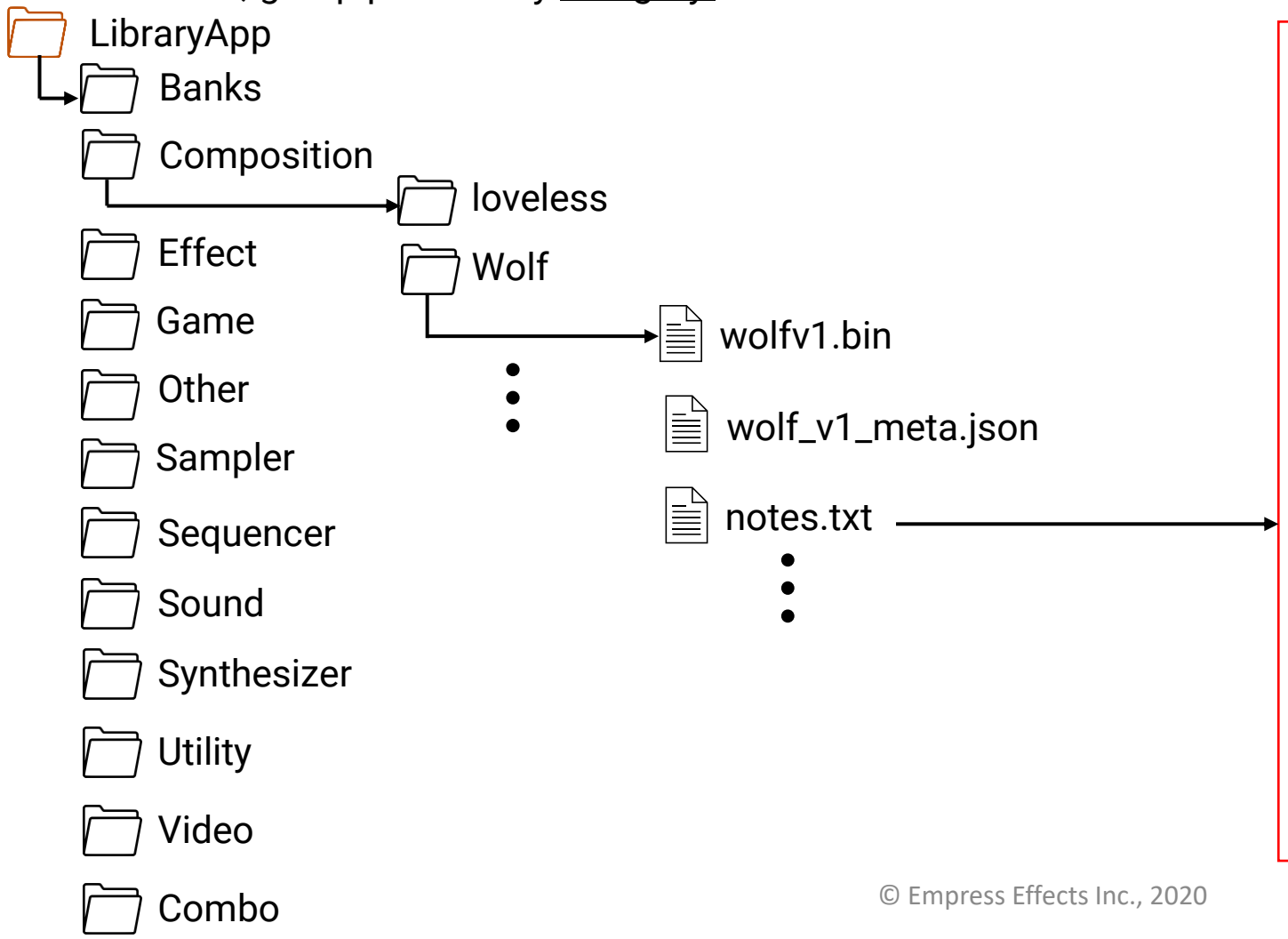
PatchStorage does not maintain previous versions of patches, they only keep the most recently uploaded files. However, version history can still be done using the Library App.

- Users can download the latest version of a patch via the Library App. If that patch is ever updated, the previous version will still be maintained locally even if the latest version is downloaded. Everything can be contained within a directory.
- An implementation could be to save the versions in the top-level directory for the patch. Then, a version identifier (v1, v2, maybe a timestamp is a better idea) is appended to each version of the patch to differentiate them.
- Specific metadata for each version should also be created in case such information changes (either a JSON or an insertion into a pre-existing database). The program will detect if a patch already exists and if it does we can prompt the user to ask if they wish to create a new version (i.e. if a new version originates from an SD card). Options to update or create a .txt file containing patch notes could also be implemented (shouldn't be difficult to implement, need to account for cases where patch notes exist on PatchStorage if the patch was uploaded as a compressed file):



Proposed Retrieval Solution

Keep it simple. Extract compressed contents (.zip, .7z, .tar, .gz, etc.) on download. Keep all .bin files intact, and store all other information retrieved as either a JSON file (which is how it is retrieved) or within a pre-existing database. If we are not going with a database, group patches by category:



If a .txt file is present with a compressed download, the app should be able to open a text editor to view, modify, and save patch notes. Could have a notes.txt file for each version(?). It would create a lot of files, probably better to condense into one file and let users format them to show specific patch notes for each version.

Proposed Retrieval Solution

Why should the patches be grouped by category?

- If a user wants to dive into the backend implementation, the organization of patches will be simple and easy to figure out.
- It will help with implementation. If a user wants to search for a specific type of patch (effect, synthesizer, sampler, etc.).
- Other options include by tag or date, which would create dozens of directories.
- Grouping by the author is also an option, but many authors create various types of patches. This would also create dozens of directories as there are many patch creators active on PatchStorage.

What happens if we go with a local database?

- There isn't a need to group patches. Tables will keep track of patches and they can be queried to retrieve the information needed. The last slide's file directory breakdown will not exist in that form. Users will only be able to access files by manually querying the database. This helps with any concerns of user-caused corruption.
- The entire folder structure would just be the Library App executable and a hidden folder containing the database.
- Since this would be a local database, the flavour of SQL that can be selected is a bit more limited.

Other options are always welcome!

Backend Storage Options

Implementation 1: Use System Defaults for Storage

Major operating systems have standards for the storage of application-related files implemented.

- The path of application files differ per OS, but there are standards that exist for macOS, Linux, Windows, etc. As long as we check the OS ahead of time, this isn't a problem.
  - Windows uses the hidden folder AppData. Of the directories contained, AppData/Roaming is a suitable option as the files could move with a user so long as they are signed in as the same user on a different computer.
  - macOS uses ~/Library/<app name> or ~/Library/Application Support.
  - Linux is iffy since there are many different OS flavours. Ideally, we make the directory hidden by using a ".", so we could use the user's home and do the following "~/.LibraryApp"

Pros

- Very straight-forward implementation. Managing files is simple in Python if given a known reference path. Organizing by patch category can help reduce sorting and retrieval times.
- Allows users that want to access the files the ability to do so easily.
- Versioning is easily accomplished using a top-level patch directory.
- Data retrieval will often be very straight-forward, not requiring advanced queries. This minimizes the need for a database when compared to storing metadata as a separate JSON file.
- The application will be smaller in size compared to one where a database is set up and used on the user's system.
- No elevated system permissions are required. The app can be run by any user and it will function correctly.

Implementation 1: Use System Defaults for Storage

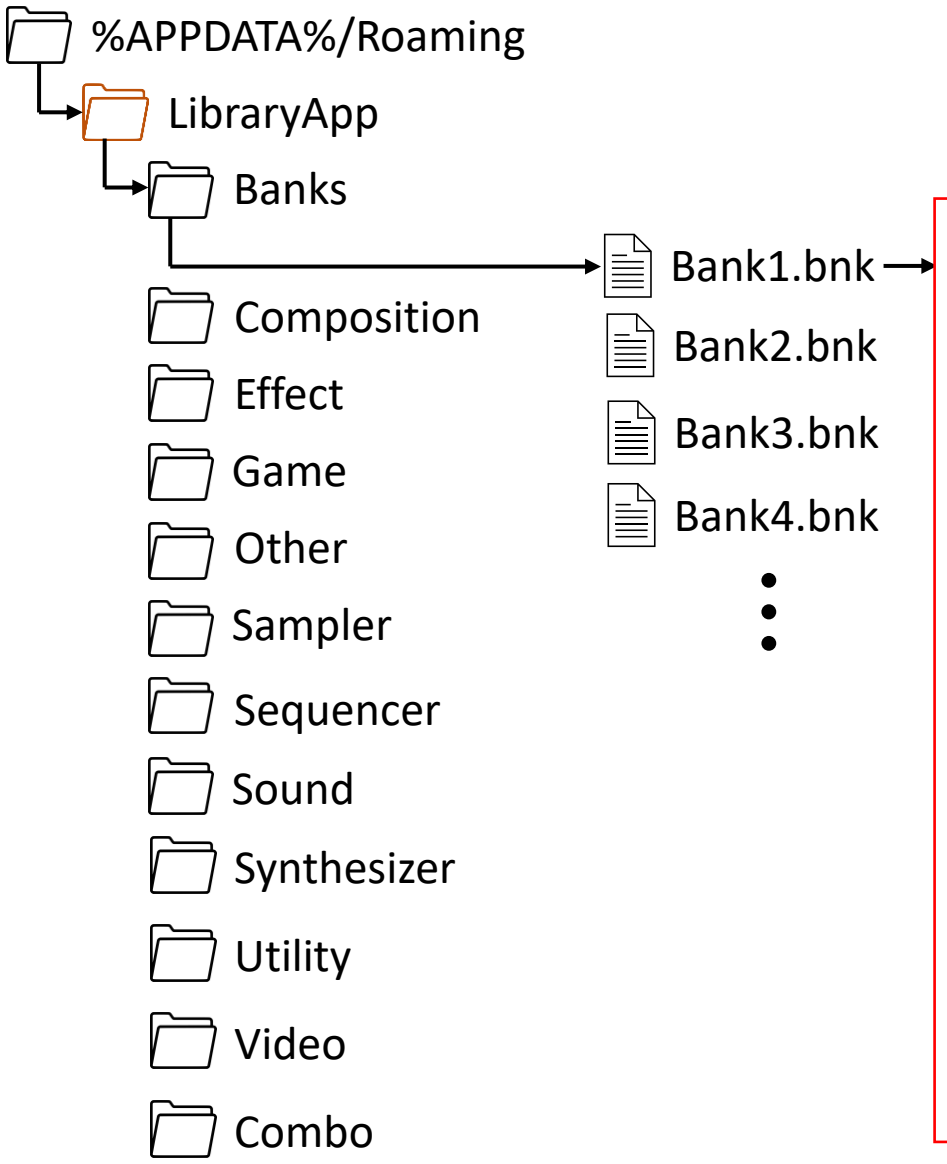
Cons

- Ideally, patches are only 1 type, but PatchStorage lets users select as many categories as they want for a patch, so the "Combo" directory may become bloated in size. This results in slower retrieval time, but given the small dataset we are working with, it won't be a problem in the foreseeable future. However, it is still something that needs to be considered to ensure it does not become a problem ever should this be the implementation chosen.
- Corruption is a real risk. Users can get into the files if they know where to look. Of course, if a user is doing things they shouldn't be there's only so much we can do to prevent it. It is still something to consider.
- Retrieval will be slower than a database implementation. Databases are designed to make retrieval as efficient as possible. Given the small dataset currently, this will not be noticed. However, as more patches are created the eventual possibility for slowdown will increase.

Notes

- Regarding bank files (as discussed in the last meeting)
  - Bank files do not need a custom format. They could easily be made using JSON or XML, but creating a custom file extension and parser wouldn't be hard to implement and gives us more flexibility about how we organize the data.
    - Users could gain access to the files outside of the LibraryApp if they wanted to see them, which could lead to corruption. That's why I would prefer a custom parser and file extension because we could check for malformed bank files without the program crashing.
  - They could easily be stored in their own directory contained within the parent LibraryApp directory.

Implementation 1: Use System Defaults for Storage




```
# Format could be “patch #:file path”. Simple, no boilerplate.
1: video/example_name_1
2: empty                          # Used to denote if a slot was not occupied
3: sampler/example_name_2
4: combo/example_name_3
5: game/example_name_4
6: synthesizer/example_name_5
```

Implementation 2: Local Database

Theoretically, this prevents the need to store anything in directories. The patches themselves, along with all necessary metadata could be stored with a local database. If this is decided as the best implementation, it is suggested to be done using **SQLite.**

Why SQLite? Do I pronounce it S Q Lite? S Q L Lite? Sequel Lite? Sequelite?

- Local databases provide many features that the LibraryApp will most likely not be taking advantage of. We do not need to make a query for patches created by a specific author on a specific day that starts with the letter ‘a’. Many flavours of SQL offer this any many more features that will bloat the size the application will occupy on a user’s machine.
- We also want to minimize the number of files a user needs to download/install to use the application. SQLite is designed to act as an embedded application database without the need for additional installations. Everything is self-contained.
- I’m not answering that last question. People have died over that discussion.

Pros

- Databases are well documented and could be maintained/changed easily without needing to know about Python.
  - It can also be mathematically proven to be in a correct form, ensuring a good design.
- Retrieval of information is very fast compared to other options. It will scale well.
- We can store anything in the database. Full files, text strings, images, etc. This allows for a large degree of flexibility.
- As long as it is done right from the start, the database schema should not need to be modified for a long time.
- Low risk of corruption since users can’t access the database outside of the application.

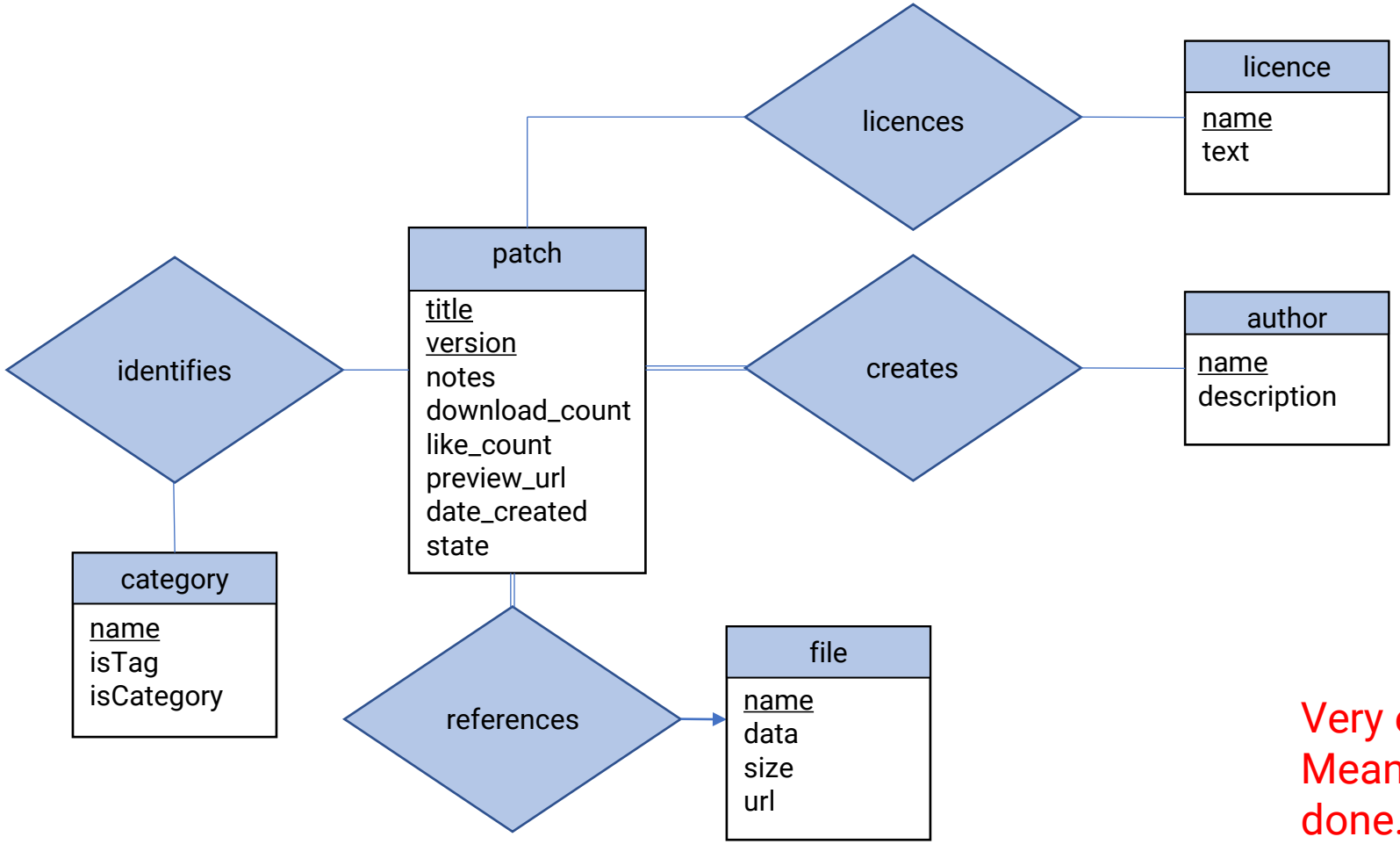
Implementation 2: Local Database

Cons

- Implementation is more challenging. On top of designing the database, we need to embed queries into the code to interface with the database. Could extend the time needed to complete the Library App by a few weeks.
- Given the nature of the application and the small amount of data we are working with, a database may be too bloated. The size of an executable with an embedded database will most likely be greater than the size of separate files on their own, as presented in Implementation 1. This also means as more patches are downloaded, the size of the executable will increase, which may worry users if they are not aware that information is being stored within the application itself.
- If the database design is not working, it will require many modifications to the code to correct the issue. It needs to be done right from the start.
- Users can't look into the backend implementation. They have no way of accessing patches and moving them around in file explorer. Good for corruption prevention, bad for transparency.

I have not looked into this enough to have a full schema ready, but here is an incomplete ER-Diagram that has not be normalized at all or had much thought put into it, but rather aims to give an idea of how such a database could be designed.

Implementation 2: Local Database

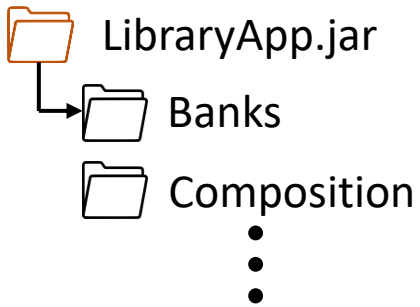


Very early work in progress. Meant to show what could be done. Databases are not scary.

Implementation 3: Forget Everything; Use Java instead of Python

No

- Horrible ideas, here for the sake of completeness.
  - Modifying files within a .jar is very ill-advised and leads to unforeseen consequences during execution.
  - What this option could offer is security against file corruption, as the files could be made hidden if they are contained within the application.
  - Of course, if the user knows a .jar can be unzipped, that doesn't matter. They can still gain access to the files and cause corruption. We could password protect the files but then users might be suspicious as to why such files would need password protection.
  - However, the positive that comes from Java is that cross-platform compatibility is a given, and if the need would arise an Android port could be completed in a short amount of time.
  - And although Java offers many database integration functionalities, so does Python.
- Despite this, a Python executable is likely the best bet. Cython could be used once the app is finished to optimize the app to ensure it runs well (also covers the case where Windows users don't have Python installed on their machines).



Proposed Backend Solution

- Given the nature and scope of the application, I do not recommend a local database for the Library App.
- Keep it simple, use the standard that other applications use (AppData/Application Support). Avoid the use of a database.
- Data can be kept secured (as the directory is hidden by default). Could optionally zip the contents and password-protect it to ensure nobody goes snooping around and corrupts the data.
- Write it in Python, compile with Cython, distribute with a mandatory terms/services agreement to ensure we are not liable if users break the licencing agreements that patch creators select on PatchStorage.
- If this is the solution, during implementation all efforts must be made to make retrievals as efficient as possible. Possibility of using a hash table to store the file path location of patches might help, but conceptually this may require extensive backend logic to implement. The custom file format/parser is a quick solution that works, but as the number of patches on PatchStorage grows this solution will only degrade in regards to efficiency.