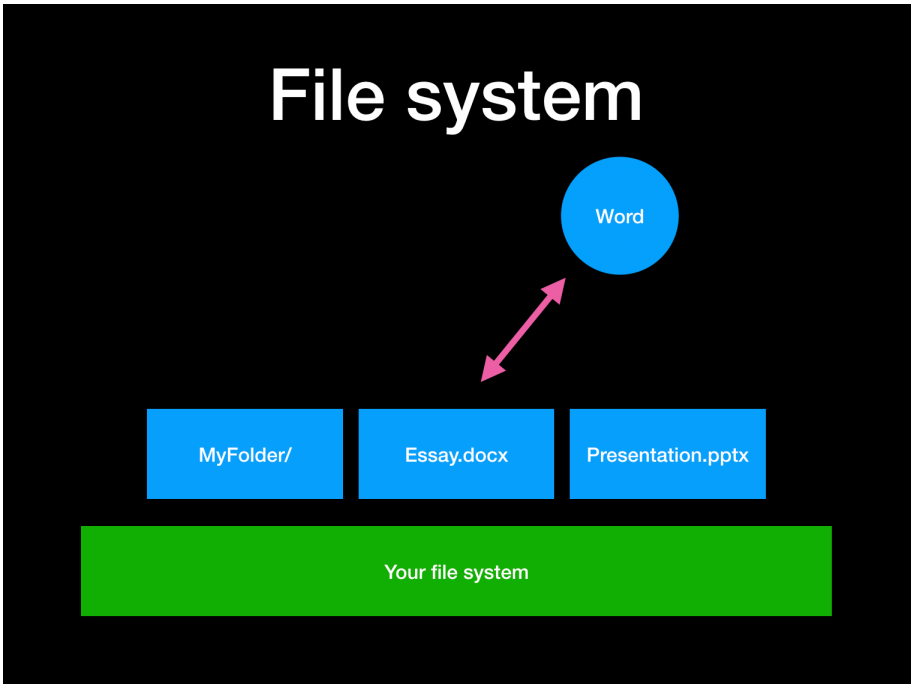
CPSC 490 - Senior Project
Sarim Abbas

Implementing a shared information atom for web applications

Defining information atoms

Much of the software we use daily is structured around units of organization and thought, which will be referred to as information atoms. UNIX operating systems (OS) use files as this fundamental unit, organized into directories (themselves a type of file). At the application level, traditional software such as word processors, graphics software, source code editors etc. also organize using files. For many years, the file has been the dominant type of information atom. Although the contents of files may vary, common OS operations (such as browsing, creating and deleting files in a file browser) are made possible with a common specification (i.e. the inode) containing metadata.

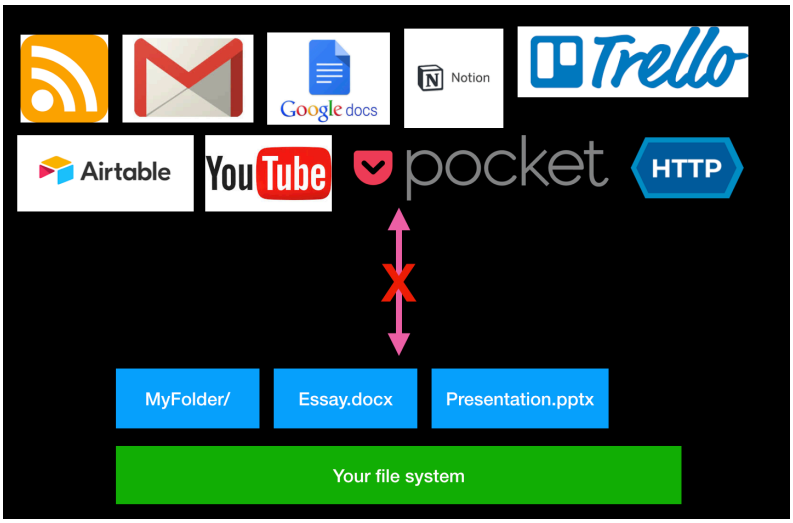


With the rise of cloud computing and web applications, we see other types of information atoms, such as emails organized in inboxes, lists of todos (Todoist, Wunderlist), cards in Kanban boards (Trello) and article feeds (RSS, Pocket). There are two points of note here. First, these atoms are more homogenous than they seem, since fundamentally they are just abstractions over lists of items (the familiar folder/file metaphor). Second, these information atoms are purely semantic and usually do not have a concrete atomic implementation, i.e. the data is stored with *proprietary* specifications in *databases* on a company's servers. Thus, the potential to exploit their similarities is lost, which is elaborated below.

Problems with siloed information atoms

The business case for proprietary information atoms is clear; we cannot expect companies to collaborate on shared atom specifications anytime soon. However, the end user suffers on three fronts:

1. Lack of interoperability: siloed atoms cannot interact with each other in context. For instance, a student may need to juggle Google Documents, a Trello board, a Gmail inbox and more apps for a school project. Instead of interacting with the atoms (which are: Docs, cards, emails) with a shared interface, they must split their work across multiple browser windows and log into each web app
2. New interfaces to learn: each web app uses its own metaphor (e.g. Kanban boards and cards), which increases cognitive load
3. Lack of ownership of data: if a web app company goes bankrupt, kills off its service or increases its pricing, the end user has limited options to archive or migrate their data elsewhere



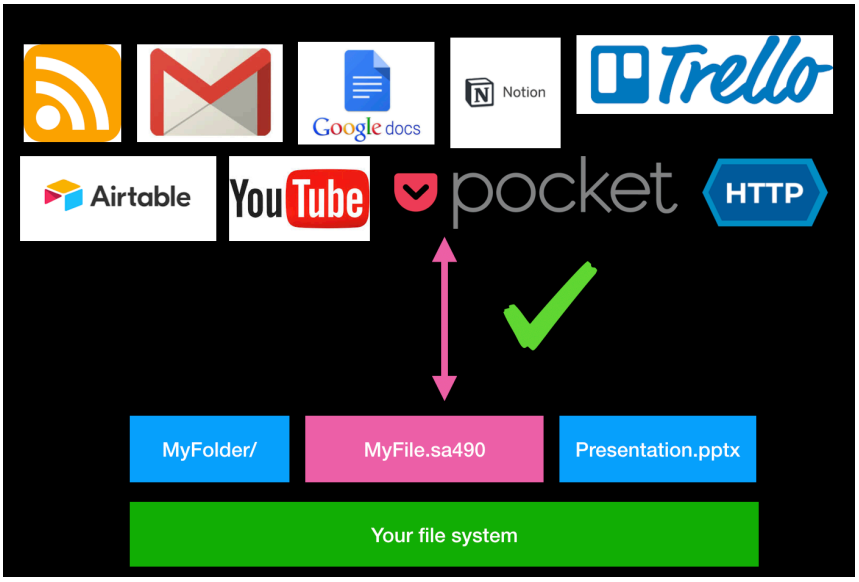
Research question

Is it possible to create a shared information atom that exploits the similarities between web applications?

Proposed implementation

My project aims to do for web apps what the OS has done for native apps with folders/files. It consists of two parts:

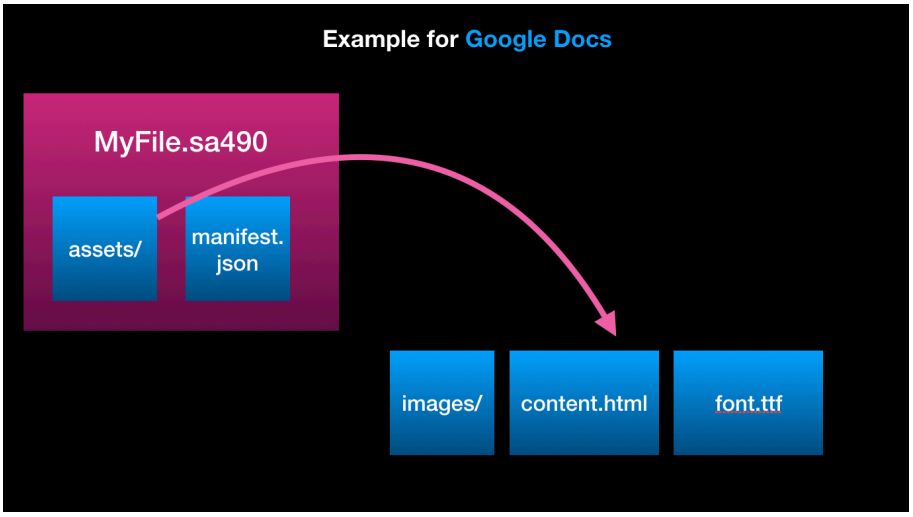
1. A common information atom for *all* web apps. This atom will be implemented as a file on the OS
2. An interface (both programmatic and graphical) to manage and modify these files, while keeping them synchronized upstream with a company's servers



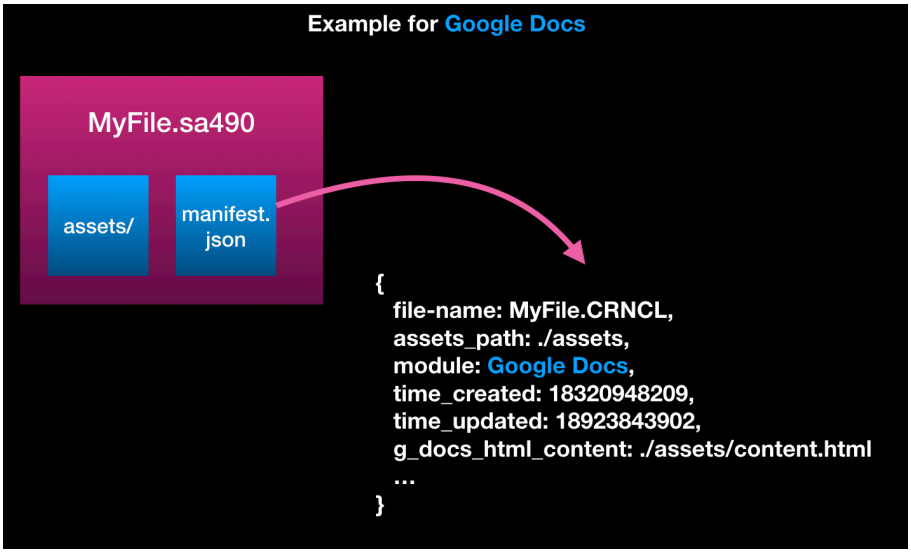
Deliverable #1: Implementation of the information atom

As mentioned, the atom will be implemented as a new type of file. Here the file is given the extension **.sa490**, although this may change.

The file will be a bundle format, similar to packages in macOS. The bundle will store all data for a web app's information atom. For instance, a Google Doc will be stored as text, along with all image assets, fonts etc. An email may be stored as HTML, along with image assets.



A JSON manifest, taking inspiration from the inode, will store metadata for the file. Due to similarities between web apps, it is expected that many of these metadata keys will be shared, opening up fresh opportunities for interoperability between files.



As a concrete example of interoperability, we might include a "deadline" timestamp in the **.sa490** specification. This immediately allows for sorting files by deadline in the interface design (detailed below). Thus, with a shared specification, we are able to treat seemingly unrelated web app data as homogenous action items.

A preliminary concern is whether other operating systems such as Windows and Linux also support package/bundle formats. A cursory analysis of other bundle formats (e.g. Textbundle) shows promise that they do. In the worst case, the **.sa490** file can be stored as a traditional archive (e.g **.zip**) and interpreted similarly by the interface.

Deliverable #2: Implementation of the interface

We can think of the interface as an augmented Finder/FileExplorer with support for managing **.sa490** files. This part of the project requires the bulk of engineering effort. At minimum, the interface must be able to:

1. Create **.sa490** files for each web app
2. Synchronize changes to files with the corresponding web app
3. Provide novel ways of organizing these files

The following hierarchy shows estimated engineering effort and the breakdown of functionality:

1. Interface
    1. Graphical Interface
        1. Sidebar
            1. File Tree: allow to open a workspace folder and recursively view the files and folders inside (effort: 2 weeks)
        2. Main Window
            1. Folder View: allow management of multiple files, with emphasis on **.sa490** files
                1. Controls (effort: 1 week)
                    1. Sorting
                        1. UNIX properties, e.g:

            1. Size
            2. Created
            3. Modified
        2. Shared properties in **.sa490** manifest, e.g:
            1. Type of web app
            2. Deadline
    2. Search
2. Views:
    1. List: similar to other file browsers (effort: 1 week)
    2. Grid: allow visualization of **.sa490** files based on any preview images specified in their manifest (effort: 1 week)
    3. Kanban: allow visualization of **.sa490** files based on any tags in their manifest (effort: 3 weeks)
3. File Creation:
    1. Create files using corresponding web app modules
    2. Allow files to be created automatically based on module preferences
2. File View:
    1. Allow modification of **.sa490** file, using module responsible for it. At minimum, this can be a web window opening the corresponding web app interface
    2. Fallback to native apps for all other files
3. Module Manager
    1. Public API: allow the community to build their own modules to interface with web apps (effort: under investigation, but expected 3 weeks)
    2. (Un)install and configure modules (effort: 1 week)
2. Programmatic Interface
    1. Background process: monitors changes to files made outside the graphical interface (for e.g. if a **.sa490** file is edited directly with a text editor), and synchronizes changes to corresponding web app servers by handing off to the modules responsible for the file

The interface requires a careful, modular approach since it has many moving parts. At present, I aim to build the interface using a JavaScript framework such as React or Vue, and will explore cross-platform solutions such as Electron, WebView or Qt. I will investigate which solution provides the right combination of flexibility and performance.

However, it is worth noting that the implementation of the interface can be done using any GUI framework and any programming language. The primary aim of this project is to advance the use of a file-based, shared information atom for web apps - the interface design is not rigid and can be iterated by anyone. The interface that I am proposing is intended only to show the possibilities of using a shared atom.

Deliverable #3: Building a system for a community

As one might intuit from the implementation, it is not possible for a single developer to build **.sa490** modules for all possible web apps. The open source community must develop these modules. Thus, the interface will have a public API. Each module is expected to provide the following:

1. A callback to handle changes made to the **.sa490** file outside the graphical interface
2. A file view component to modify the **.sa490** file within the graphical interface

3. A service to create **.sa490** files manually or automatically (for e.g. an RSS module may automatically create files for articles as they become available from a publisher). The files created are expected to share as many keys in the manifest as possible

There are at least two approaches to integrate these modules into the interface. The first is to create a marketplace (similar to the extension stores in IDEs like VSCode) where anyone can publish their code, and users of the interface can choose the modules they want to download and enable in a dedicated Module Manager. This poses a number of challenges, including:

- Security and auditing of published modules
- The need to "hot-reload"/recompile the interface to initialize new modules
- Providing a smaller, clean public API that isolates the module's code. For a web-based GUI, this would involve encapsulating CSS styles, JS code, use of the global store etc.

Another approach is to fully integrate module development into the Git master. In other words, active members of the open source community will vet and build official modules. External additions will be submitted via pull request and approved. The immediate benefits are a less complex architecture, since there is no need to recompile the running interface, and modules can more easily hook into the existing code without a well-encapsulated, versioned API. However, this comes with its own challenges:

- Each module is delivered as part of a release. Users will have to manually update their interface (or OTA, which comes with its own implementation challenges) to receive module updates
- A potentially higher barrier to entry, since hobbyists/hackers cannot put together their own unofficial extensions for services of their choice
- Community debates about which features are worth implementing for each module

Although I will attempt to investigate both approaches, I will make a decision based on both ease-of-implementation and ease-of-use for a community.

Illustrative proof-of-concept

In the fall semester, I created a standalone note-taking app. It relies on a **.sa490**-like file format. It can be viewed here: https://github.com/sarimabbas/intrepid. The project helped me learn how to create desktop GUIs. I hope to integrate this as a module into my senior project.

Related work in this area

Many have tried to wrangle diverse app data before, but their attempts end up as augmented web browsers. Franz, Rambox, and Station are examples of interfaces that hope to create harmony by bringing common web apps together in tabs. At most, they provide unified search, but ultimately the different information atoms are still siloed from each other.