Rory Byrne

COMP0004 Coursework Report

Features

My Java Web Application implements a simple way to store, search, and view collections of notes. The user interface (UI) focuses on being clean and intuitive, designed as a retro-style website with minimal styling (although it takes a lot of effort in styling to achieve the minimal look). This stripped-down design keeps the focus on the application's core functionality, ensuring the user experience remains straightforward and distraction free.

Notes are stored in a hierarchical folder system, my interpretation of requirement 6, where notes and folders can be created inside other folders.

These notes can be created, edited, renamed, pinned, moved between folders, and deleted. They can contain text, images, and URLs via an interface inspired by Jupyter Notebook where the user can create, edit, delete, and move around interweaved blocks of different types.

My web app is composed of 3 main pages, the **home** page, the **my notes** page, and the **search** page. The **home** page shows you a list of your pinned notes and the notes you have most recently modified. The **my notes** page allows you to navigate through your notes and folders via a table with sortable columns for their name, when they were created, and when they were last modified. This page also features breadcrumb navigation, allowing easy folder traversal. The **search** page allows you to search for any note by its name or its content. Clicking on a note on any of these pages takes you to the display page for that note where you can see the contents of the note and access its edit page. All data is stored in a JSON file and automatically updated when changes are made.

Design Process

At the beginning of development, I spent a lot of time deciding on the overall design of the app, the classes, and how the data will be stored.

I implemented the folder structure by creating Folder and Note classes that share common functionality through inheriting from the abstract Item class. This abstract class forces its subclasses to have IDs, names, and timestamps as attributes as well as methods like generateId, toJson, and a comparison used for sorting. For folders the ID is generated using the name of the folder, this is so that the folder path is readable when displayed in the URL (which I will talk more about later), and for notes this is generated using milliseconds since Unix epoch, this is so that each note is generated with a unique ID as it is not possible to create two notes in the same millisecond through the app interface. Folders aggregate a map of Items, associating each item's ID with the item itself, allowing for O(1) lookup when given a path of IDs. Using a map means it does not maintain order but this is not a problem as the order is applied when retrieving the folder's contents. The entire hierarchical folder system is stored in Model as a Folder itself, named root.

For the notes themselves, I really wanted to allow them to have interwoven contents of different types, so for example you could have multiple images anywhere in the note with text around them. This eventually led me to the idea of the Jupyter Notebook inspired design with different blocks that can be created, edited, deleted, and moved around. I implemented this by creating classes for each type of block that inherit from the abstract Block class. This abstract class forces its subclasses to

have an ID as an attribute as well as methods like generateId, toJson, and search. The block IDs are generated using the same technique as notes. Notes have a list of blocks. All lists are realised with ArrayList as the number of items is so small there is no worry about efficiency.

All the data is stored in a JSON file. I decided to use a JSON file as it is simple, human-readable, and easy to parse using Java libraries. The structure of the JSON file preserves the hierarchical folder structure where each folder is a map of IDs to items (which can be further folders). The JSON file starts with the 'root' folder and also stores all attributes like timestamps and names. Blocks are stored in the file in note items as a list of maps in order to preserve their order, containing their type, content, and ID.

These decisions and using good object-oriented design principles have made my application much more flexible for any future development, meaning minimal refactoring is required for things like creating new items for folders and creating new blocks for notes.

The note editing page was challenging to create. In terms of its UI, I had to make the blocks as intuitive and interactive as possible and keep the design in the same theme as my other pages, a very iterative process. This same iterative process was used for the UI of every JSP page. The EditNoteServlet has both a GET and POST method, allowing the editNote JSP to retrieve data from the model as well as update it. Folders and notes are created and deleted using servlets with GET requests as it allows these performed by simple URL links that then immediately redirect back to the page they came from. Since these actions don't involve sensitive data, the lack of security in GET requests isn't a concern. Also, this allows the actions to be performed within notes themselves using URL blocks, which is pretty cool.

When I created my folders and notes, I made it a priority to have the URLs of their pages to show the path that they are in. This means the design of my URLs follows the same hierarchical structure using items' IDs, providing a clean and organised way to access folders and notes and simplifying server-side processing since the path in the URL (that can be easily retrieved) directly maps to the folder structure. This URL pattern is used for displayFolder, displayNote, and editNote JSPs. Furthermore, I use parameters in the URLs to specify the current sort with the attribute I'm sorting by and the direction, allowing me to preserve sorting throughout navigation of the folder structure in an intuitive way.

I followed the Model-View-Controller design pattern for clear separation of concerns: JSP pages handle the user interface, Servlets manage requests and responses, and the Model handles data management using JSON. The use of a Singleton pattern ensured a single instance of the Model class for consistent data access. I have used plenty of abstraction and polymorphism, simplifying lots of operations, and the classes I have created are cohesive, reducing complexity by each being responsible for one clear purpose.

I believe I have implemented and designed my app well, meeting the requirements. I feel the minimal-styling highlights the efficiency and effectiveness of my implementation, and I am very proud of what I managed to make.