COMP0004 Web App Coursework Report

Overview of the project

The user of the web app is greeted with the welcome page. They can choose to select one of the 3 functionalities from the menu bar, which are: viewing the complete list of all existing notes, creating a new note and searching for a note.

When viewing the complete list of notes, the user can choose to:

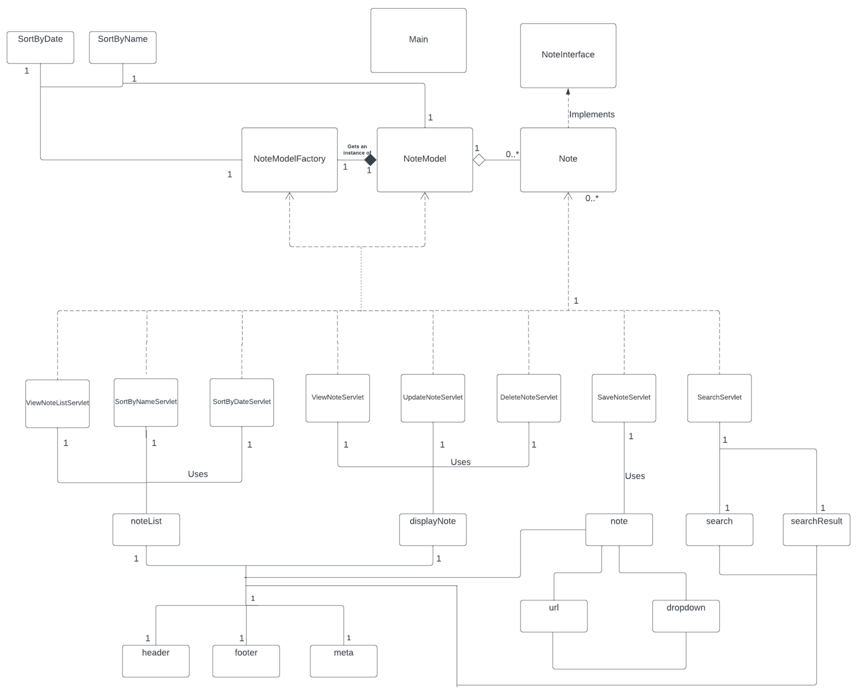
1. View any individual note
2. Sort the notes by alphabetical order
3. Sort the notes by date created with the newest appearing first
4. Sort the notes by date last edited with the newest appearing first

When choosing to the view a specific note, the user views all the properties of the note in textboxes which they can edit. They also get a list of URLs found in the note which are clickable and redirect them to the correct resource. Once they are done editing the note, they can simply click the update button and the file is saved automatically. Otherwise, they can also click the delete button, and the program automatically does so.

When choosing to create a new note, the user can set a title to the note, set the index of the note (which basically resembles a folder structure), and enter the content of the note. Users can use a dropdown menu to select plain text or a URL to a resource or image. However, they can even include a mix of plain text and URL and the program will automatically recognise them.

When choosing to search for a note, the user keys in the search keyword which can be anything from the title, index or even the content. The search is case insensitive and removes unnecessary spaces and returns all the corresponding matches. The user can view the results list, view any individual note, and perform all the same tasks as when viewing a note.

UML Class Diagram



Design process and analysis

To start, I made a list of the features I wanted to include in the project. As guided by the specification, the Model View Controller (MVC) design pattern was respected and implemented. The Jakarta Servlet Pages (JSPs) would make up the view, the servlet classes would be the controller and the Java classes would be the model.

Bearing in mind that notes must be stored, I then considered the possible storage alternatives to store the notes. JSON proved quite appealing as it allowed me to create a single JSON object, containing a JSON array, which in turn contains JSON objects representing the notes. Having planned the basics out, I reviewed the design thoroughly ensuring that everything was cohesive.

I started the implementation by making the JSPs for the pages I'd want to display. These JSPs require data from the back end, that is from the JSON file. JSPs themselves cannot fetch that data, so I then created specialised servlet classes, which communicate with JSP files using HTTP requests. These servlet classes are responsible to get required data which matches their purpose by calling methods from the model side of this MVC design pattern.

During the implementation stage, I came up with the idea of making an abstraction for the notes. By making use of the 'Note' class to represent note objects, a good use of abstraction was made. Instead of always handling primitive data types which would require a lot of manual tracking, the 'Note' objects are instead very convenient to use and understand. When designing the 'Note' class, I first made a 'NoteInterface' interface which helped ensure that no methods were left out in the 'Note' class. Following that, I made my 'Note' class, which implements the 'NoteInterface' interface. This supports good Object-Oriented Programming (OOP) practice.

The note class itself comprises instance variables representing the note properties, a constructor and a series of getter and setter methods. These note objects are then held in an ArrayList, which acts as an index holding all the notes. This allows for easier and faster access as the JSON file does not need to be accessed every single time a note has to be handled. The ArrayList is created once at the start of the program, so it can simply be accessed and edited as changes occur on the notes.

Further adhering to good OOP practice, a good use of encapsulation was used throughout the project by making sure that the auxiliary methods used are made private instead of public. For example, the setter method and the clear file method used as helpers in the NoteModel class are made private because they do not need to be used by other classes. Besides, all methods in servlet classes were made 'protected' so that their scope is limited to the servlet package only. This practice promotes security.

As for the model side, the model package holds classes that make up the backbone of the whole project. The NoteModelFactory gets a single instance of the NoteModel class, which is then used throughout the running of the program. The classes found in model are responsible for automating the processes that the user wants to do. For example, reading from and writing to the JSON file.

Overall quality of work

Aside from adhering to good OOP principles, I instinctively made use of good programming principles and conventions. Firstly, I tried my best to limit the scope of variables and methods. I also made use of meaningful variable names and Camel Case naming convention to improve readability and ease of debugging.
Secondly, I made reusable code in some classes to reduce redundancy and duplication. For example, I used the same class for sorting by date created and date edited but selected the correct attribute using the ternary operator. Redundancy was also reduced in JSPs as the recurring menu bar on top of the page was written once in the header file and simply included in other JSPs.
Finally, try-catch blocks were included to handle any possible errors occurring.

Throughout the implementation, care was taken to create a cohesive, efficient, and clean set of classes, that meet all the criteria expected. A widespread design pattern, MVC, was respected making the project match industry standards.