Production Tool Proposal

Joseph Orme

Dansk Scanning

<u>Internal Use Only</u>

Table of Contents

Executive Summary

The production tool and production in general is struggling with a main and key mechanism with regards to scanning. The tools are outdated and are starting to provide more problems and solutions and are in need of a change. To try and fix what is there would be costly and require months of work to understand what the tools are doing and what improvements could be made upon them. The categorization tools suite is a hurried mess comprised of several flavors of SQL databases, all trying to communicate with each other and trying to synchronize with each other in an effort to keep data up to date and the databases ‘clean’. There is a clearly better solution available with regards to the tools that would cut down on maintenance, reduce complexity, and provide production with a greater throughput of work if applied correctly across the entire production tools suite.

This proposal suggests the following approach: one singular shared database that is able to help and meet the demands of both production and the management in regards to information needed to store, maintain, remove, and modify. The shared database will allow all the tools to work off of the same pool in the correct order and prevent any need for separate databases for each tool. This new approach will require a new design, proposed solutions and of course ideas for examples. This proposal attempts a best effort to solve these issues based on management requirements, software requirements, and available techniques and ideas at the time of writing.

The readers of this document are free to accept, refuse, refute, or question any decisions they wish, the purpose of this proposal is to argue to management from a place of experience, research, design, and collaboration to provide a good and clear solution that not only solves the current issues but tries its best to prevent future issues.

Problem Overview

The production consists mainly of scanning sensitive documents, brought to us by the customer in boxes. These boxes are then scanned, every single document into production where they are then sent to the archive process. Each document is signified a category, and each page that is part of the document is also signified a category (hereto within this proposal will be referred to as Document Category and Page Category respectively).

There are two main, and significant problems that occur in Production

1. Customers do not store all documents of the same category in the same box. When a box arrives, it could contain one, two, three, or many documents that are part of different archives for the customer. While there may be a logic to the way they are stored inside the box, documents in one box could be related to documents in another box, being worked on by several different production workers. These documents need to be able to be related to the same grouping (referred to here as the Archive) but a box is processed on an individual basis. This leads to production needing to reconcile after everything has been scanned what goes where and which belongs to whom.
2. Customers are picky at best, indecisive at worst when it comes to their document categories. Document categories are arranged by the type of document, and grouped by the type of archive being processed. Customers have different naming conventions for each archive, document, and per page. Some might have more than one naming convention for the same type of document, due to language or some other policy. Each time this occurs, it is a tremendous slog on the current tool and requires manual DB changes and modifications to keep things clean and accurate.

Changes like these aren't impossible to take with the current tool, but it's a significant problem with the way things are. The current production tool for categorization is based on the following:

- A 'prep' database, that is a SQL database that maintains the raw records from the production scan. So far this is rather stable and provides little to no hassle.
    - However the use of SQL makes the rest of the system inflexible, as everything is tied in some form or fashion to the prep database's relations. Some tools don't or will have 0 need for all of the relational tables in the prep database
- A 'tool' database, using MySQL that is then attached to a GUI using Java. The database calls for the tables in the prep database, then used them to map the various Page Categories, Document Categories by the assigned or requested Global IDs for the pages scanned in Prep, and then relating those to the customer and their approved strings.
    - This 'synchronization' of the two databases is a lot of overhead and provides plenty of places where one Db could mess up the other
    - This is hindered also by the fact that the DBs communicate with each other to try and keep in sync, but this can lead to a situation where one database gets 'dirty' by the other one. It happens a lot more than one might think.

- A ‘post-production’ tool that is dependant on the other two tools, then prepares the files scanned to be sent to WebLager to be available to the customer.
    - This tool is using a PostgresSQL database to keep track of the files ready to be added to the queue, and their respective categories assigned to them.
    - Another potential problem with the databases needing so many different sync operations, although this one is the safest being the one at the ‘end of the line’ with regards to production operations.

There is vast room for improvement in regards for how things work in this environment. For the start of things, we will focus on the side that focuses more on the categorization of things prior to be sending to post production.

Description of Problem

The problem description is more helpfully described in the form of requirements. The requirements for production to work better is listed in as best detail as follows:

- A user is able to create new categories for a particular customer
- A user is also able to edit and modify existing categories for a specific customer
    - An edit to a string for one customer shouldn’t necessarily change it for everyone else. Several customers use similar categories so to change one for one customer should not change the category for other customers using the same category.
    - As well, customers should be allowed to have ‘custom’ categories and as such, free to add what categories they wish, specific to themselves and themselves alone.
- A user can change archives and process the same documents
    - This is because once again, sometimes customers do not provide all the documents related to one particular archive in one box, and sometimes across multiple boxes
    - The idea being that the tool can handle things on a box-by-box basis rather than a archive-by-archive basis
- A user should be allowed to remove, or add existing categories to their list.
    - A removal of a category should result in it being removed from the customer’s list of usable categories. If its a common category used by many customers than only that particular customer should lose it, and it should be usable by other customers still.
    - If the category is a unique one for the customer, then it might be safe to remove entirely from the system.
- A user should be able to modify an archive when done or remove it from the system
    - Archives when done with should be cleanly and safely removed from the database
    - This can apply to other parts of the database like documents, boxes, or even whole customers themselves.

Description of Tool Proposal

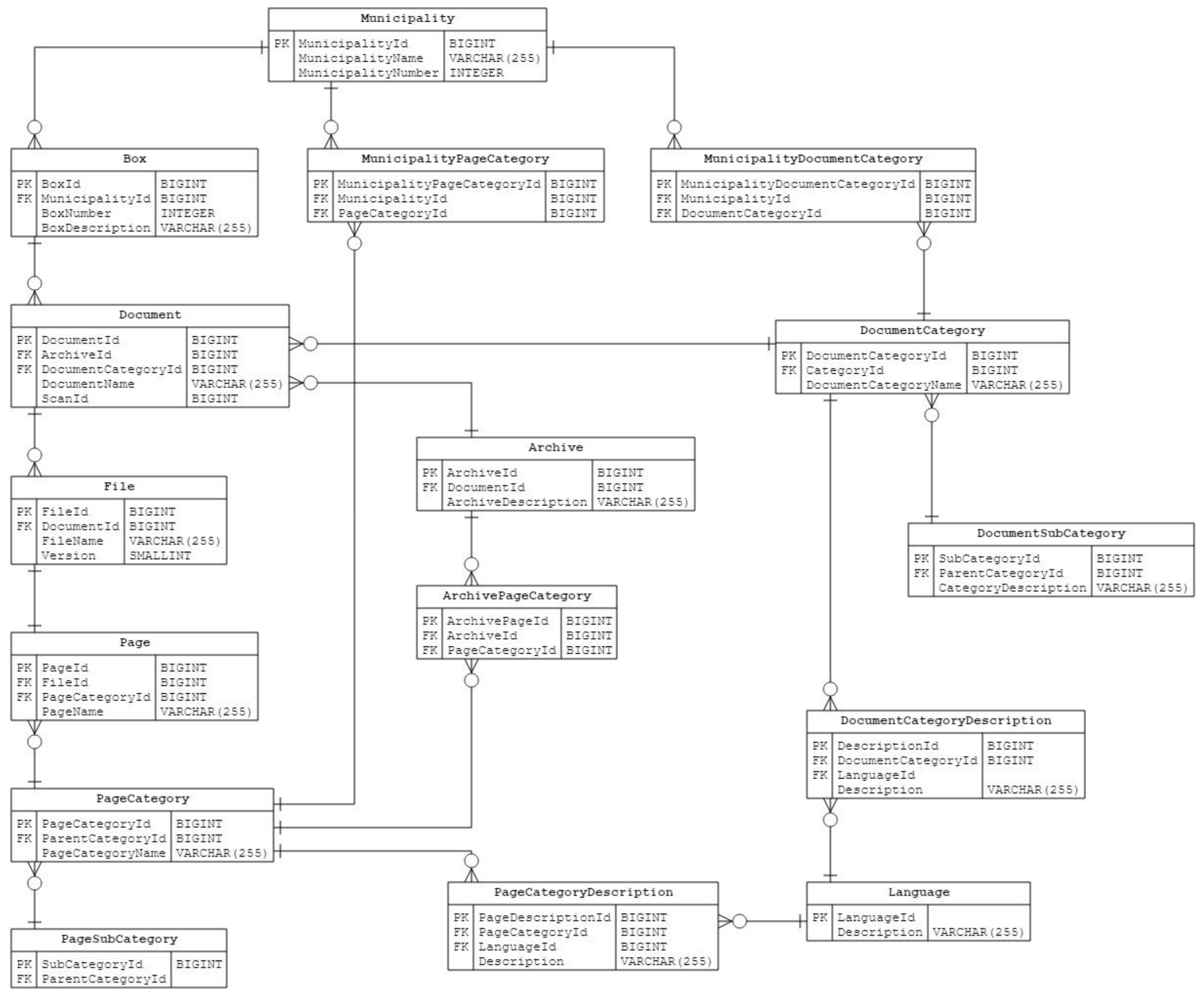
The tool should be described in the following way

- Using Java 8
    - Needs to be an open source tool per management
    - This requires a sort of GUI as well for desktop applications, JXML has been proposed, need further analysis for this.
- MongoDB
    - Proposed as the database backing tool and persistent store, to be used with regards for its scalability and ability to make a less complex schema
- The tool consists of one persistent and scalable database, capable of handing the many changes that will occur due to use and increased user needs.
- When the user wants to use the tool a typical process would be like as follows
    - User selects a customer
    - The user can then select an existing archive or create a new one
    - The user would see a list of files production has scanned and be able to assign the following to each file
        - The associated document and document category
            - This category is one that is listed as available for the customer/archive.
        - The associated files and their respective page categories
            - These categories are listed as available for the customer/archive
    - Each Document would then be saved and stored in the database with the correct information
    - This information is then used by post-production to send the newly categorized file to WebLager
- This tool would allow for future design and scale, thanks to a database designed for handling changes easier than the current production tool

Database Overview

Schema Considerations

The following is a schema proposal that was given by management, created first in Microsoft Access and then later translated into a UML Database diagram using Star UML:



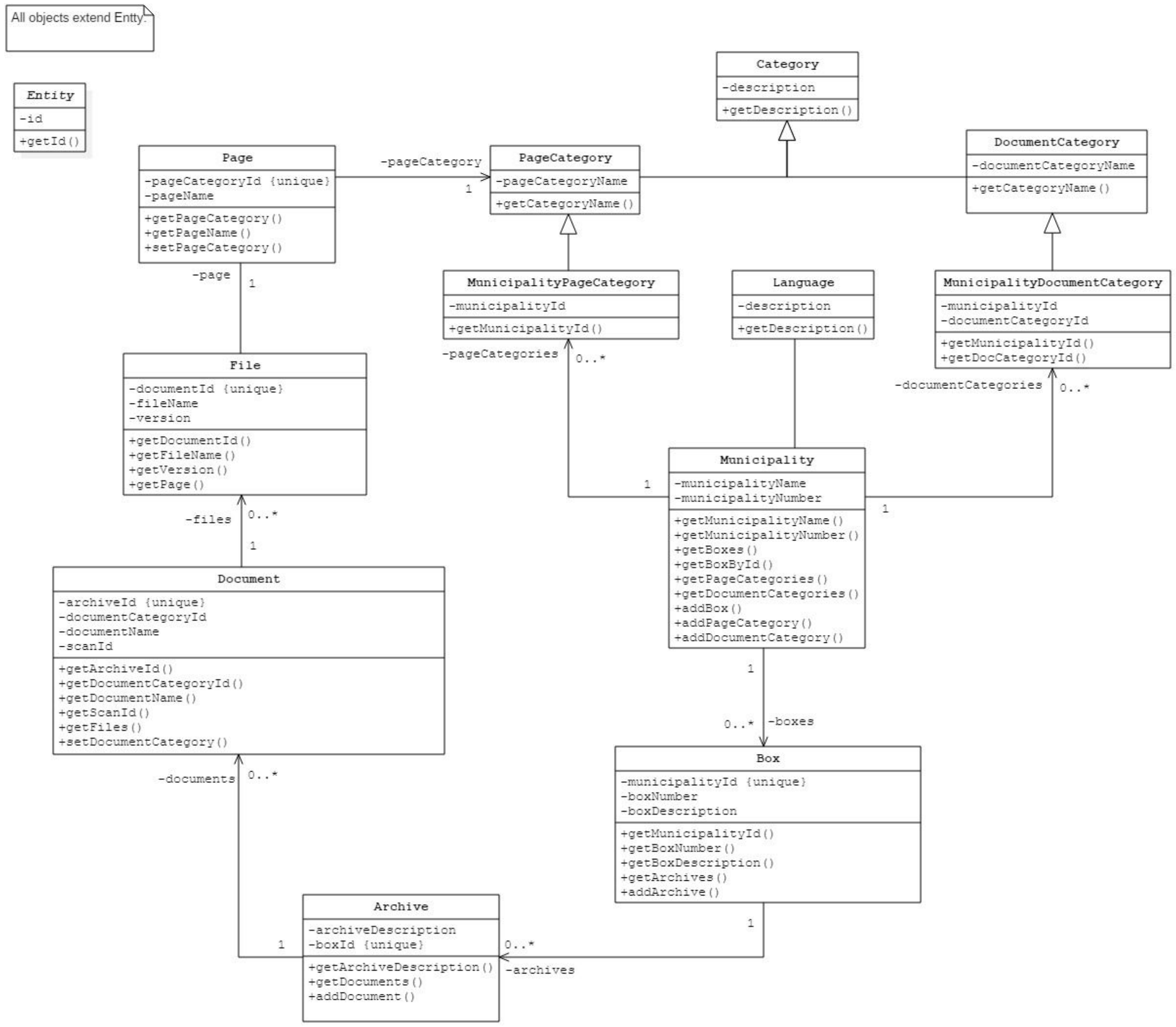
There are a few caveats and problems with the proposed schema:

- It is a relational model, which makes it very rigid and hard to change or adapt. Granted that most of these tables are unlikely to change columns rapidly, but several would and should be anticipated to change a lot, particularly the “xxxCategory” tables.
- Language is tied to the category, when in reality the category itself isn’t what changes, its the customer using said category. This relation would probably be more directly tied to the

customer(or Municipality) table that way, any customer has any string tied to their specific language, rather than having a multitude of possible categories with languages tied to them.

- Several of these tables seem redundant, and would only serve to make things more obscure and complicate the code that calls down to it. While the distinction might make sense in a class diagram it seems to make less sense in a database schema
- The categories for documents seems to take a wide turn in the schema. Instead of be directly related to a category, it needs to reach the Page Categories through the Document table.

A proposed class diagram of the schema, revised and refined is as follows:



A few notes and considerations and concerns:

- **Considerations**
    - There is a reduction in tables with this schema. Instead of having a seperate table for descriptions, all categories regardless of whether they are for Documents or Pages are all able to provide a description of themselves.

  - The ‘Municipality<X>Category’ could likely be reduced as well. At the time though, the idea would be to separate categories that are across the board, from categories that are visible to the municipality. This would also allow for the consideration of a municipality creating a ‘custom’ category for themselves.
  - The Document is related to the MunicipalityDocumentCategory, although it cannot directly be associated to it via a dependency line. The category a Document recieves is passed to is via the relation to its Archive, which in turn is related to its Municipality and the categories available.
  - A Page is related to a single PageCategory, which in turn shares a one-to-one relationship with the File
  - The reason for File? A Document has the potential by a customer to be changed somehow. When a customer changes a Document, instead of just overwriting the existing files we create a new collection of them called a Version internally. Why this is I don’t know nor necessarily agree with, and in this case causes added complexity to both the schema and the classes used to talk to the DB.
- **Concerns**
  - One future concern is how new values or classes or entities will be added to the database and the corresponding tool. Perhaps a metric for how many categories needs to be added to the tool. Management uses a tool that scans the prep database and uses it as a measurement for worker productivity, could that be added or utilize the same database? If so, would it need a new table under the database, and would said table need to be able to communicate with this one.
  - Another concern is how much more complicated things get if the data in each class gets out of control. Right now, there is a few parameters in each class with simple getters and setters for methods. Some of the fields are not used at all in the current proposal, but were artifacts of the db schema. Should we removed these and worry about adding them back later?

MongoDB

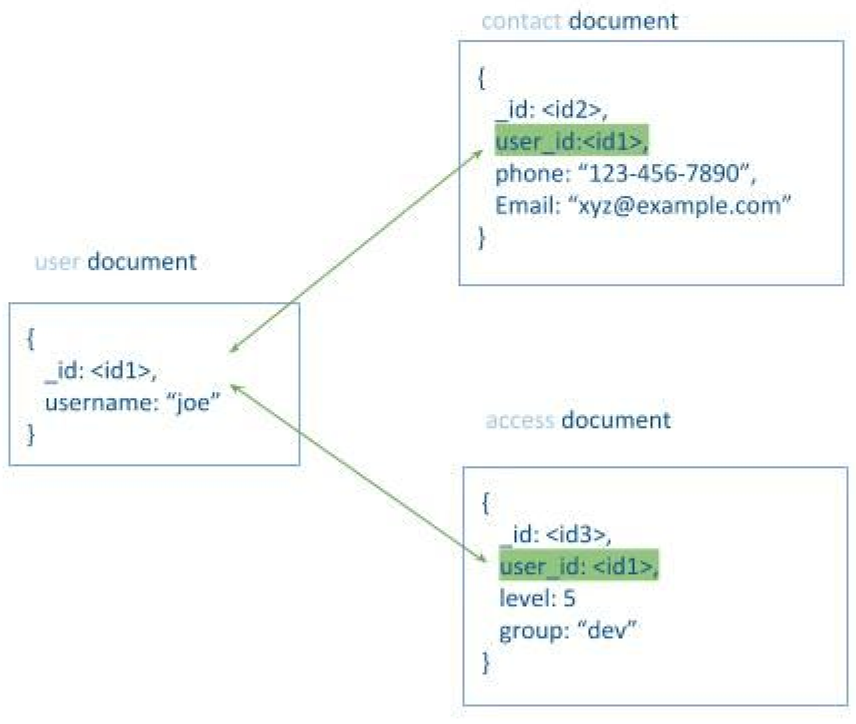
MongoDB is being proposed as the main driver of the collective database. MongoDB is an open-source, NoSQL database language that applies the following caveats to its structure.

Everything is a Document

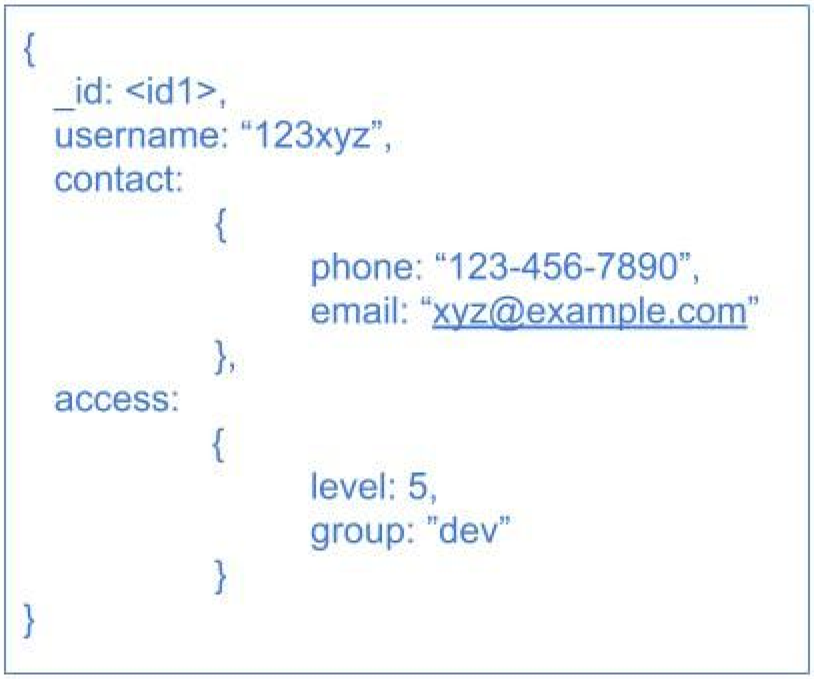
MongoDB treats all records at the heart of the database as a structure they label a **document**. Documents are simple JSON/BSON objects of key/value pairs like below:

```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

Documents can be related to each other in two main ways, either we can use a traditional reference based schema:



Or you can ‘embed’ related data into the object



```
{
  _id: <id1>,
  username: “123xyz”,
  contact:
            {
                  phone: “123-456-7890”,
                  email: “xyz@example.com”
            },
  access:
            {
                  level: 5,
                  group: ”dev”
            }
}
```

In general MongoDB recommends the following:

- Use Embedded Data Models when
  - You have a “contains” relationship, or a 1-to-1 relationship between entities
  - You have “one to many” relationships between entities, and the entities are always views with regards to each other (in our cases, we’ll never look at Documents alone, they would need to be viewed with regards to their Page and Files. Files and Pages are a clear 1-to-1 relationship in our schema)
- Use “Normalized” data models when
  - Embedding would result in duplication of data that doesn’t outweigh the read performance advantages
  - Many-to-many relations are part of the schema
  - Dealing with large hierarchical data sets (for our situation, categories are a clear hierarchical data structure)

It is likely that the production tool should likely use a combination of the two, which is possible with the proper planning of the data structures. For example. Files and Pages are 1-to-1, which could be embedded. Embedding these files in their respective documents also seems doable. For archives and their respective Documents, you can either embed or go to a more normalized model. Categories could be hierarchical, with relations normalized between the various Documents and Pages, with references stored in the municipality entities to help with reducing duplication.

Key Features

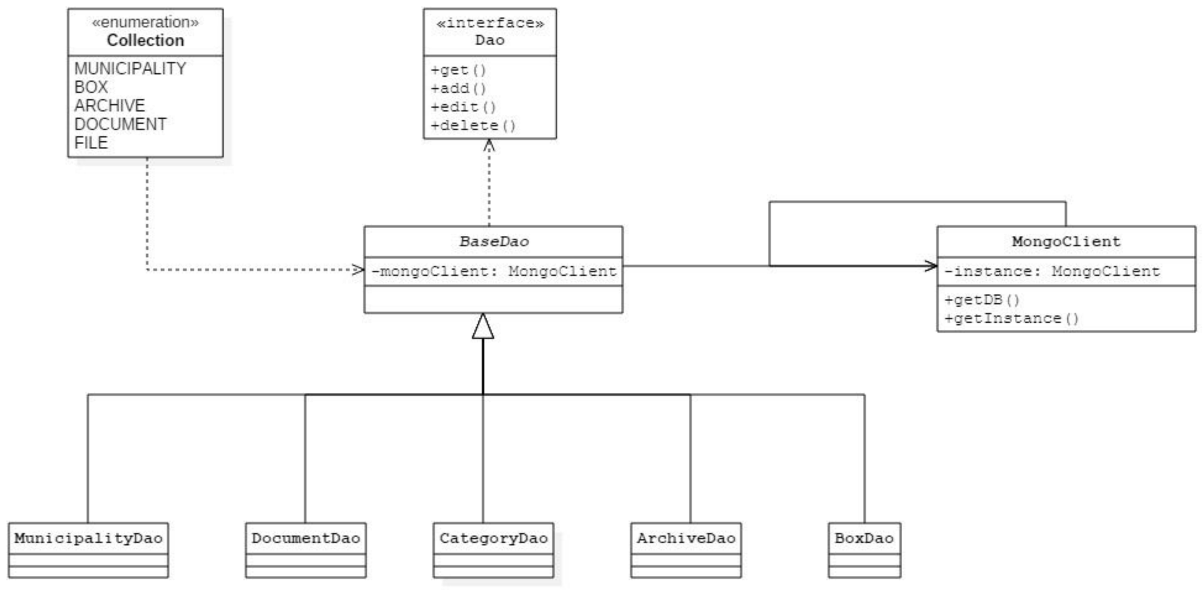
- **High Performance**
    - MongoDB using documents allows a multitude of languages to talk to the simplistic nature of the records with ease
    - The JSON/BSON structure allows for the use of embedded documents, which helps reduce the need for joins or other expensive operations in a more traditional database.
    - Support for embedded data models and indexes increase query speed and reduce I/O activity.
- **Flexibility**
    - The rich query language provided by MongoDB supports all standard CRUD operations
        - Also allows for text searching and geospatial queries.
    - MongoDB provides a mechanic called the replication set, which allows and provides support for automatic failover and data redundancy
    - MongoDB is scalable *horizontally* which allows for sharding and provides help for clusters if needed.

Given the scalability , flexibility, and the rather simplistic nature of the tool and the needs of the managers, MongoDB is a solid recommendation to help with regards to help with a DB that can be managed, queried, indexed, rolled over, and anything else we need while allowing for a simple and robust structure.

Class Overview

The following diagrams are to help with regards to the proposed structure of the backend code, including use cases and sequences.

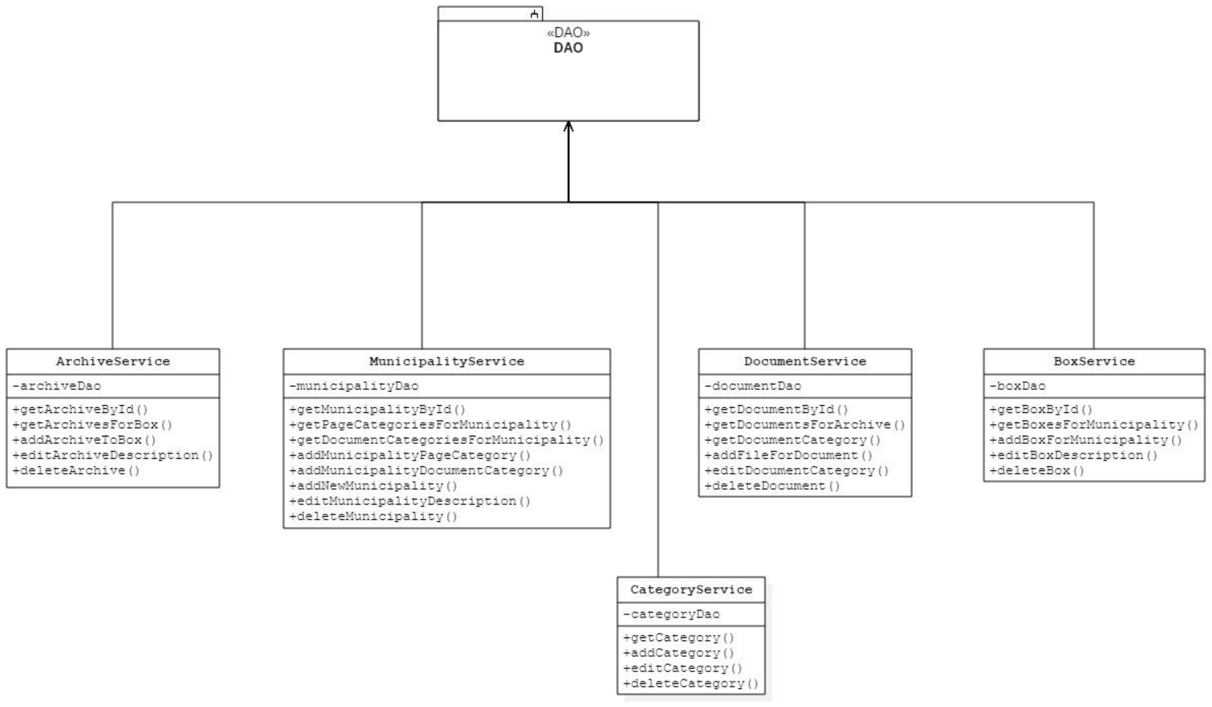
Dao Class Diagram



<u>Considerations</u>

- MongoClient is attempting to use a Singleton pattern, eliminating the need for multiple instances of the client that talks to the database and the various collections (MongoDB's answer to SQLs tables)
- BaseDao is the main caller to the db, which specialized in the standard CRUD database operations.
- All other child classes are able to specialize on the various operations or the specified fields for each entity in the related collection
- Collection enumeration is supposed to help with regards to the caller. It might be better to be a part of the individual DAO classes, but the BaseDao making the call would do better with regards to the proper place to call. For example, there could be a simple *get* call to the municipality database like follows:
  - `get(MUNICIPALITY, m_id);`

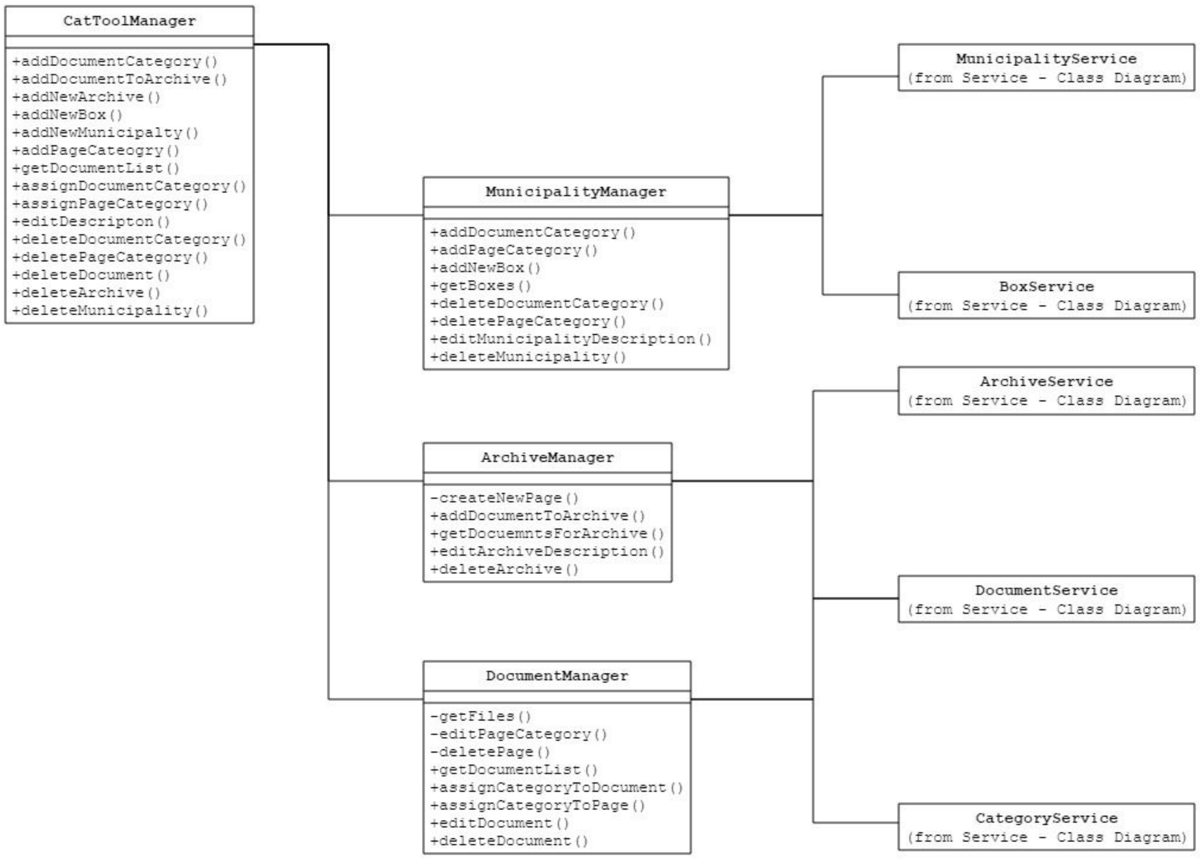
Service Class Diagram



<u>Considerations</u>

- Service layer contains a bit more business logic, the mid-layer between the database and the front end GUI for the tool.
- There is no File or Page Service, this is because of the consideration that a document would have the Files and their respective Pages embedded into the Document
  - This means the Document Service needs to have a bit more business logic, although most of it is rather simple with regards to changing something embedded inside of it.

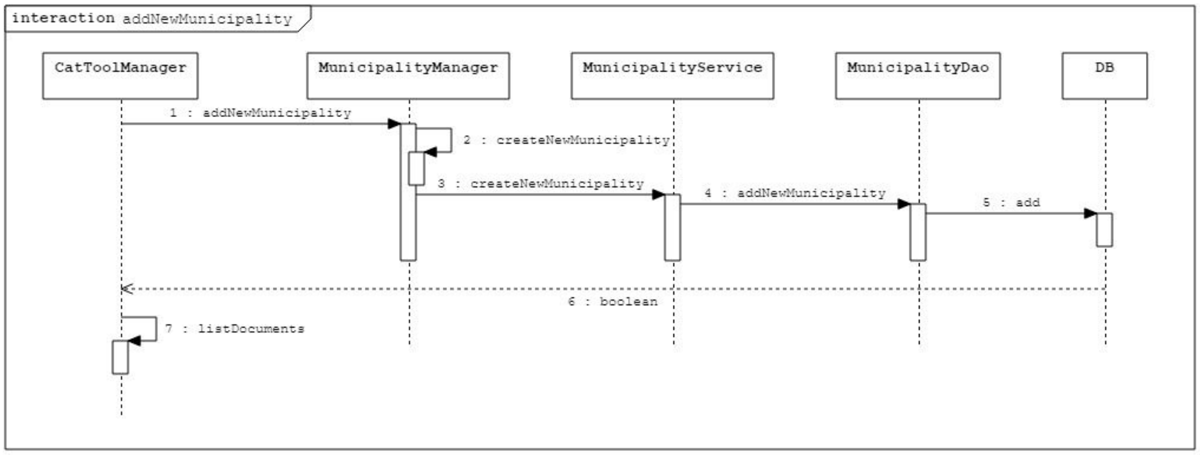
Manager Class Diagram



Considerations

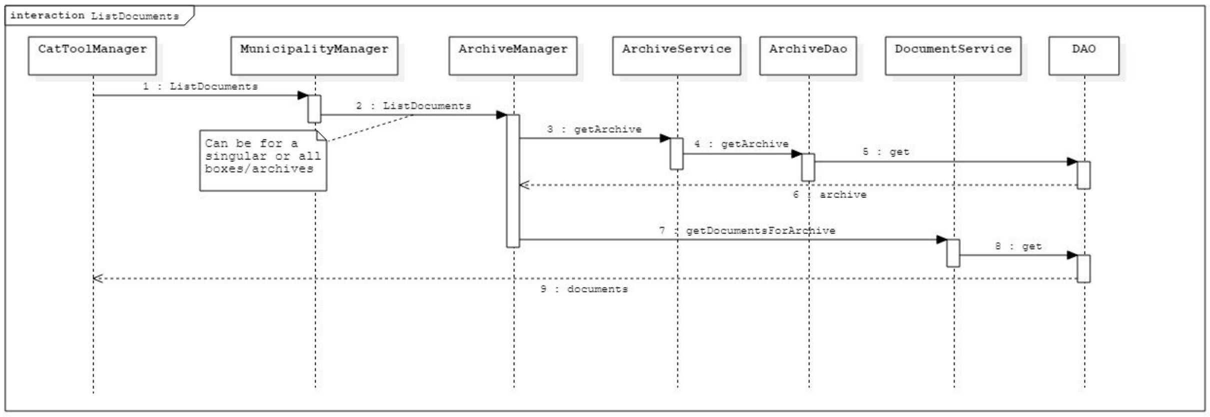
- The Manager layer is derived from a Facade Pattern. The CatToolManager faces the GUI and is comprised only of calls to sub managers who handle the actual work.
- The managers are spaced out with the following justifications
    - The Municipality manager only needs to know how to find municipalities, and to manager the various 'boxes' that are assigned to the municipality.
    - Any box that is found can be handled by its archives by a different manager, this municipality manager should only care about returning the box and then it can be used by the Archive Manager.
    - The Archive Manager can see both the Archive Service and the Document Service, this allows it to add documents related to archives and grab the list of documents from the appropriate archive.
    - The Document Manager is in charge of all other operations, mainly tasked with the categorization of both Documents and the various pages within. Hence it needs to be able to call the Category Service for the purposes of grabbing the right ones. Typically, this is handled by the municipality categories as well, which should be passed down to it from the results of calls to the Municipality Manager.

Sequences



Adding a new Municipality

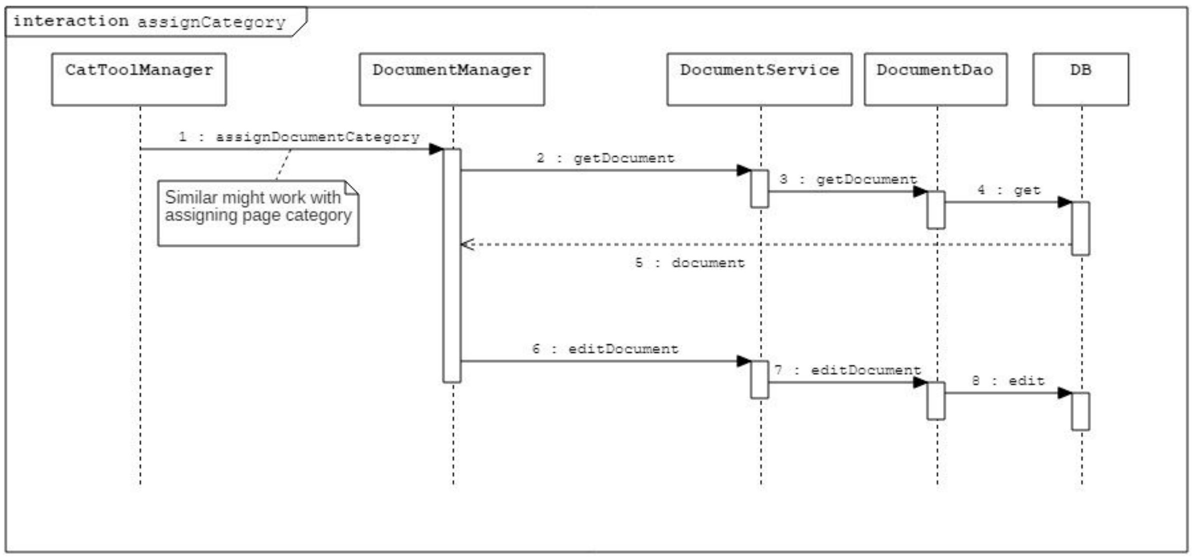
- The CatTool is called, the GUI likely will have fields that are necessary for the creation of the archive by default (like name, description, etc).
- The Manager will assemble the information provided into the Municipality entity
- The Municipality Service calls its relative DAO, which sends the DB the JSON/BSON formatted Municipalty entity
- (Optional) the GUI could refresh itself and show the new empty Municipality.



Listing documents for a municipality

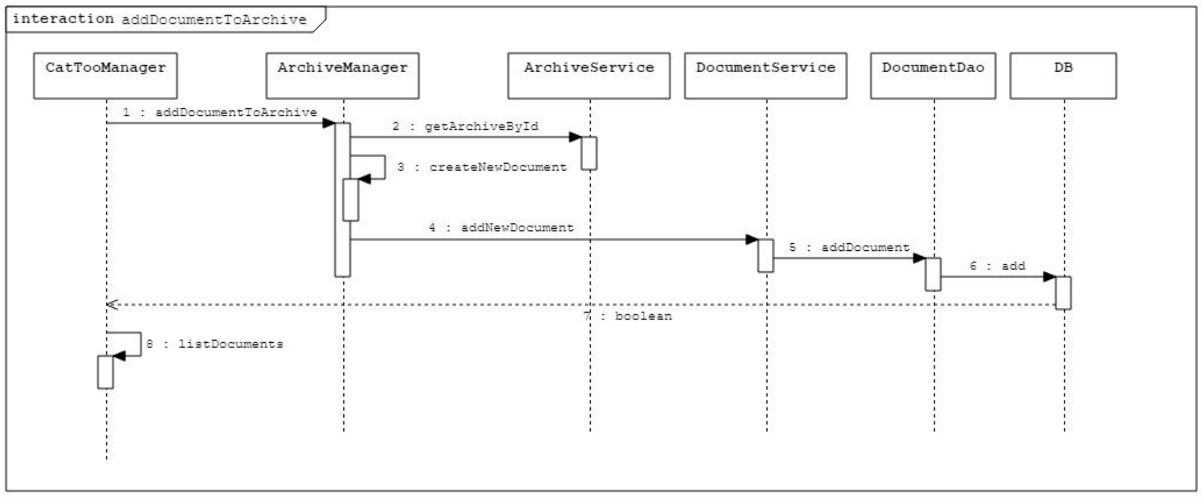
- The CatTool fetches the appropriate Municipality from the Manager, which in turn calls the Archive Manager with the archive
- The Archive Manager grabs the relevant archive
    - This could be a multi-archive operation theorhetically. In that case, steps 3-8 are in a O(n) loop

- The Archive Manager makes the call to the Document Service that allows for the GET to the database.
- The list of Documents are returned to the GUI



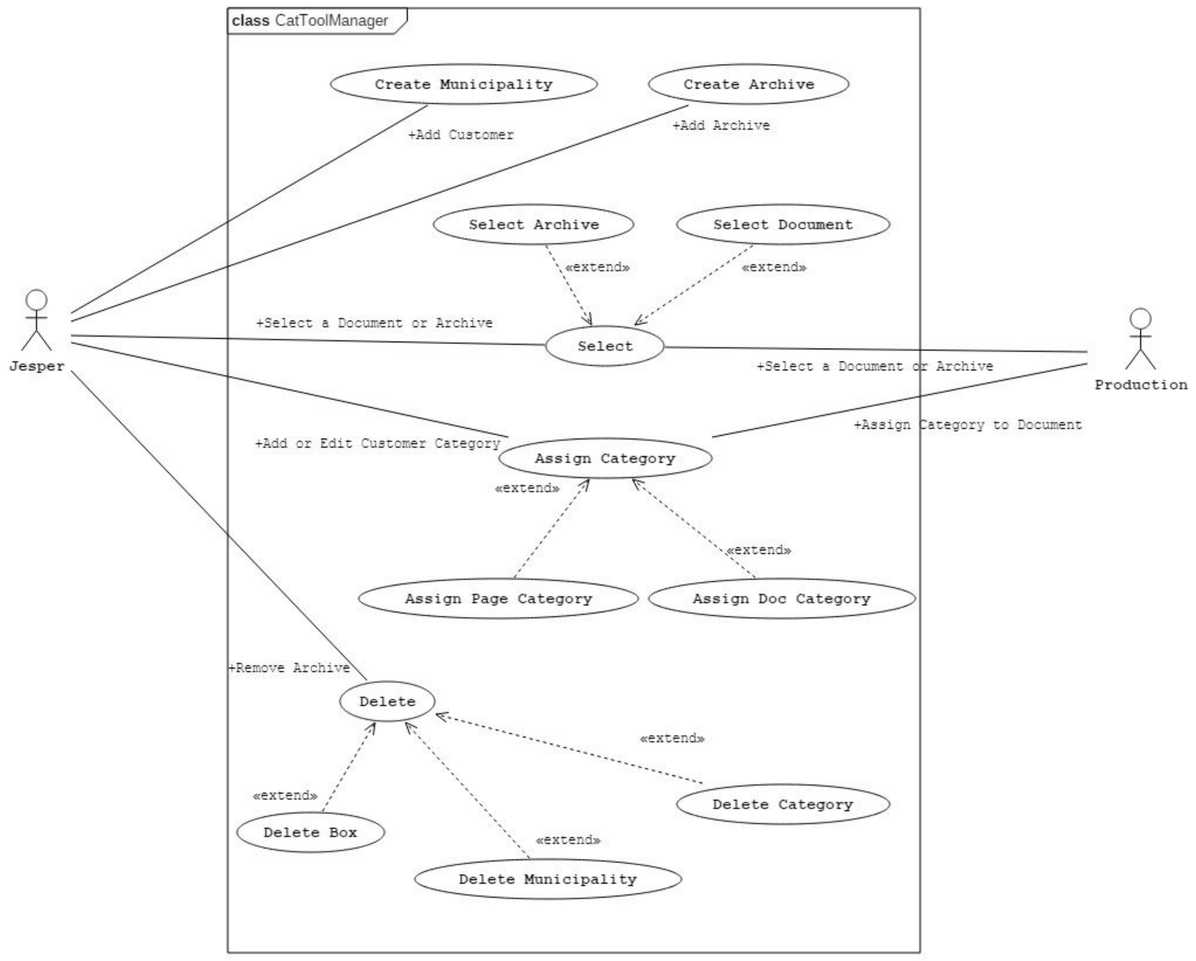
<u>Assign a Category to a Document or Page</u>

- The CatTool is given the appropriate category, and the document to assign it to
    - This would likely be in terms of an ID that is assigned to the document and then the category ID.
- The Document Manager retrieves the document, via calls down to the DAO and the database.
- The Document entity is edited to assign the category to the document
    - This same operation could be used in a similar fashion and sequence for the Page that is associated to the Document. In this case, the list of files would be looped through O(n) times to find the appropriate File with the corresponding FileID and PageID.
- The Document is edited and sent to the DB for the write
    - There could be error cases involving an improper write
    - As an option the GUI could refresh and show the newly assigned category, or it could notify the user of a successful write.



<u>Adding a new Document to an Archive</u>

- The CatTool is offered a new Document that has been scanned
    - This might be done in a separate process through the prep database.
- The Archive Manager grabs the appropriate Archive the Document is assigned to, and then creates the new Document Entity
- The Document Service calls down to the DAO and database to make the write
- (optional) A boolean is returned to the CatTool Manager to inform of the successful write
- The List is refreshed to show the new Document to the list



<u>Use Cases</u>

- For the Admins
    - An admin needs to be able to see and look at all the various parts of the system, including all standard CRUD operations
    - The admins use cases involve the creation and modification of categories and archives that are used by Production for later
    - An admin can modify categories, both for documents and pages per municipality
    - When a category is not being used, the admin can remove the municipality, archive, or category itself.
- For production
    - The production worker should be able to see and select an archive and a municipality, with a list of static documents that have been scanned
    - When a production user assigns a Document, it is assigned to the appropriate archive, municipality, and categories
    - The Production user can't remove anything, they are only allowed to see and 'assign' values to Pages or Documents.