Data-Repository-App Documentation

Prepared by Mike Brown

January 2016

[Introduction 3](#_Toc440470279)

[User Documentation 4](#_Toc440470280)

[Upload Data screen 4](#_Toc440470281)

[Source Document 4](#_Toc440470282)

[Attachments 5](#_Toc440470283)

[Data Category 5](#_Toc440470284)

[Find Data screen 5](#_Toc440470285)

[Features accessed via a REST API 9](#_Toc440470286)

[REST Endpoint: Removing a “Dataset” 9](#_Toc440470287)

[Background 9](#_Toc440470288)

[Usage Details 9](#_Toc440470289)

[REST Endpoint: Adding a new Data Category 9](#_Toc440470290)

[Background 9](#_Toc440470291)

[Usage Details 10](#_Toc440470292)

[REST Endpoint: Repopulating the Database 10](#_Toc440470293)

[Background 10](#_Toc440470294)

[Usage Details 10](#_Toc440470295)

[Developer Documentation 11](#_Toc440470296)

[Database 11](#_Toc440470297)

[The Document Store 14](#_Toc440470298)

[Java Code 15](#_Toc440470299)

[Source Code Organization 15](#_Toc440470300)

[User Interface Code 17](#_Toc440470301)

[Application Configuration 18](#_Toc440470302)

[Multipart File Settings 18](#_Toc440470303)

[Security 19](#_Toc440470304)

[Admin Documentation 20](#_Toc440470305)

[Installing MongoDB 20](#_Toc440470306)

[Starting MongoDB 20](#_Toc440470307)

[Stopping MongoDB 20](#_Toc440470308)

[Mongo Best Practices 20](#_Toc440470309)

[Memory Settings 21](#_Toc440470310)

# Introduction

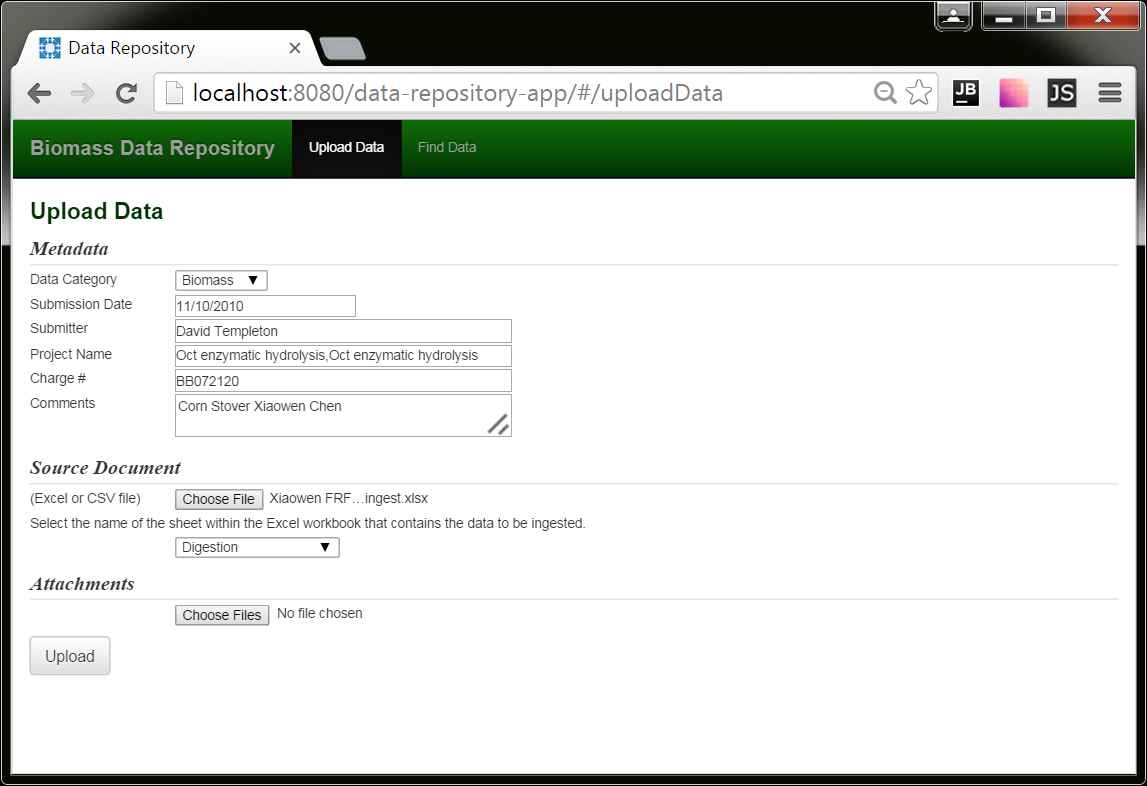
The Data Repository App has been developed for the purpose of storing grid-shaped data (i.e. rows and columns) and retrieving it later. It has a browser based UI that users can use to upload data in the as Excel workbooks or comma-separated-value (CSV) files. The ingested data is stored in a database. The raw uploaded files are stored on the server’s file system. The app provides a mechanism for searching for data rows using user-defined criteria as well as the ability to download the originally uploaded files.

The initial design and development work on this was done by Mike Brown (mike.public@superbrown.com) for the three month period from October through December of 2015.

# User Documentation

The user interface consists of two screens, one for uploading data, one for finding data (i.e. searching for data).

## Upload Data screen



When the user uploads data to the system, he also fills out “metadata” fields specifying the following:

* data category (required)
* submission date (required)
* submitter (required)
* project
* charge code
* comments

Besides being retrievable, metadata fields can be used in search criteria.

When data is uploaded or searched upon, it is done so within the context of a particular “data category.” Data categories provide a mechanism for segregating data of different natures from one another, essentially allowing them to behave as separate data stores.

### Source Document

If the source document (the one containing the data to be ingested) is an Excel workbook, a selection list (as pictured above) will appear containing the list of sheets contained within the workbook. The user will use this list to select the sheet that contains the data the user wants the app to ingest.

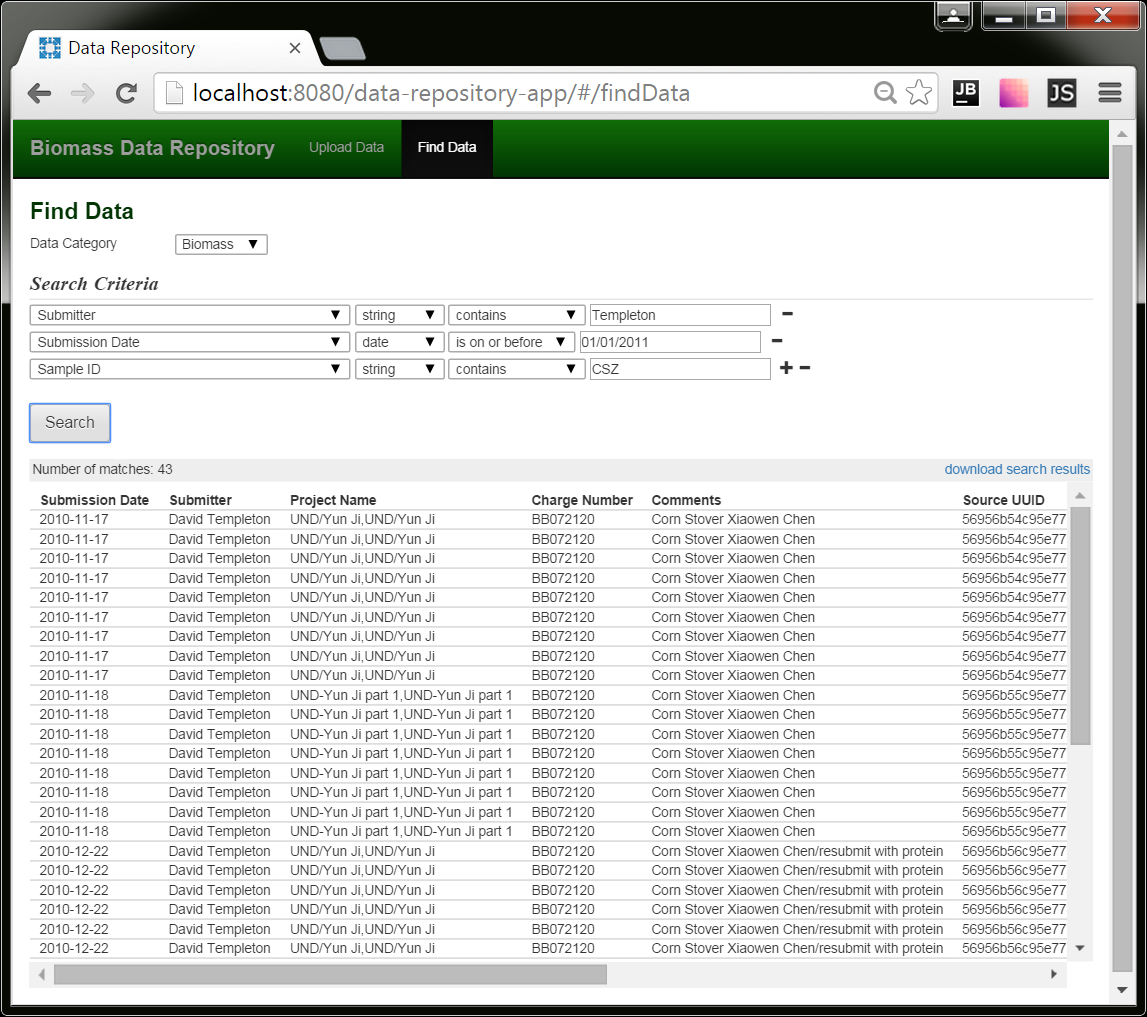
### Attachments

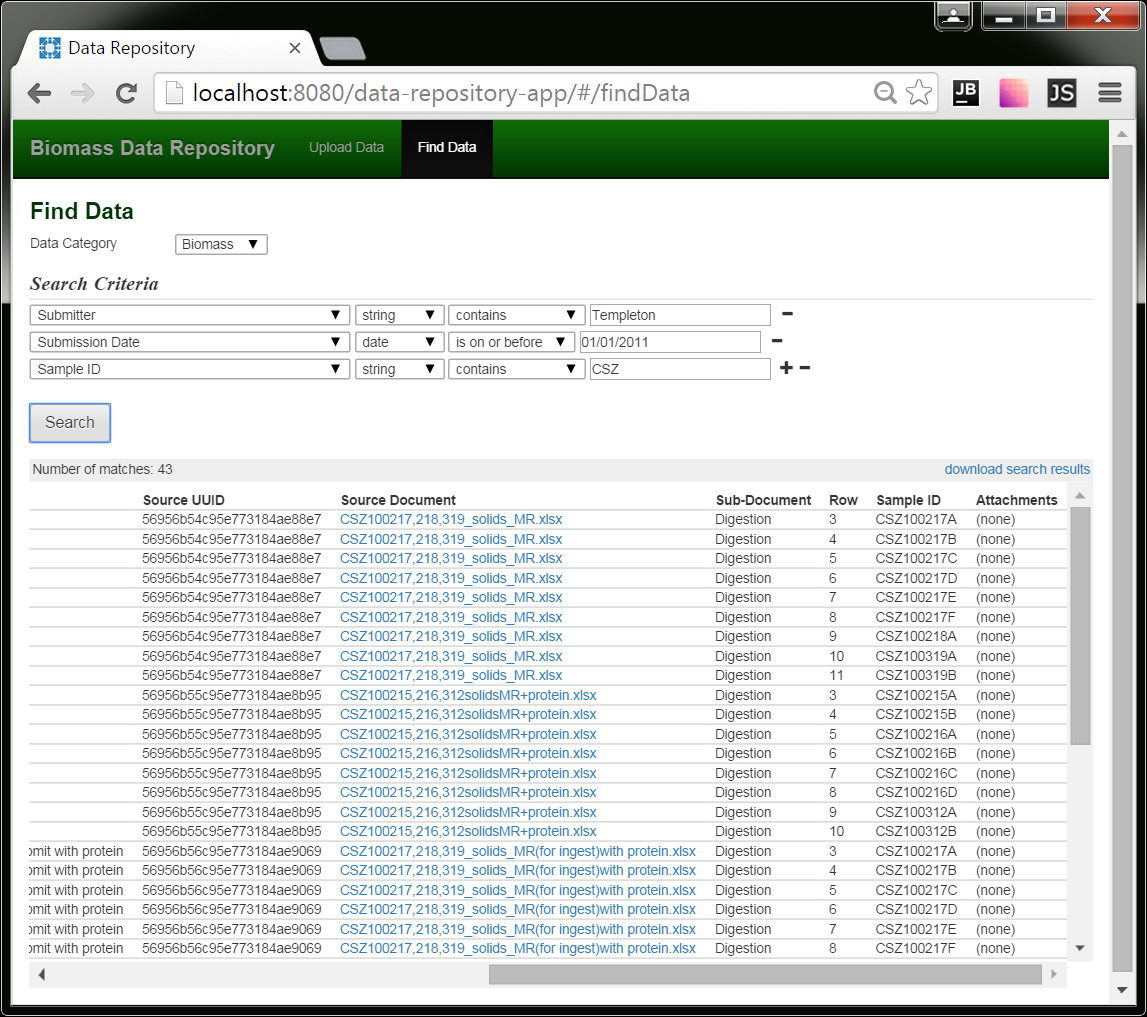
The user can also include, in the upload, files he considers “attachments.” These are retrievable later.

### Data Category

The data category is selected from a drop down list and is data driven. Data categories are added to the system in one of two ways. The first is to add it to a comma separated setting in the application’s configuration file called app.defaultSetOfDataCategories. When the application launches, it assures these categories exist in the database. The second way of adding them is to call a REST service, which is explained in the admin documentation.

## Find Data screen



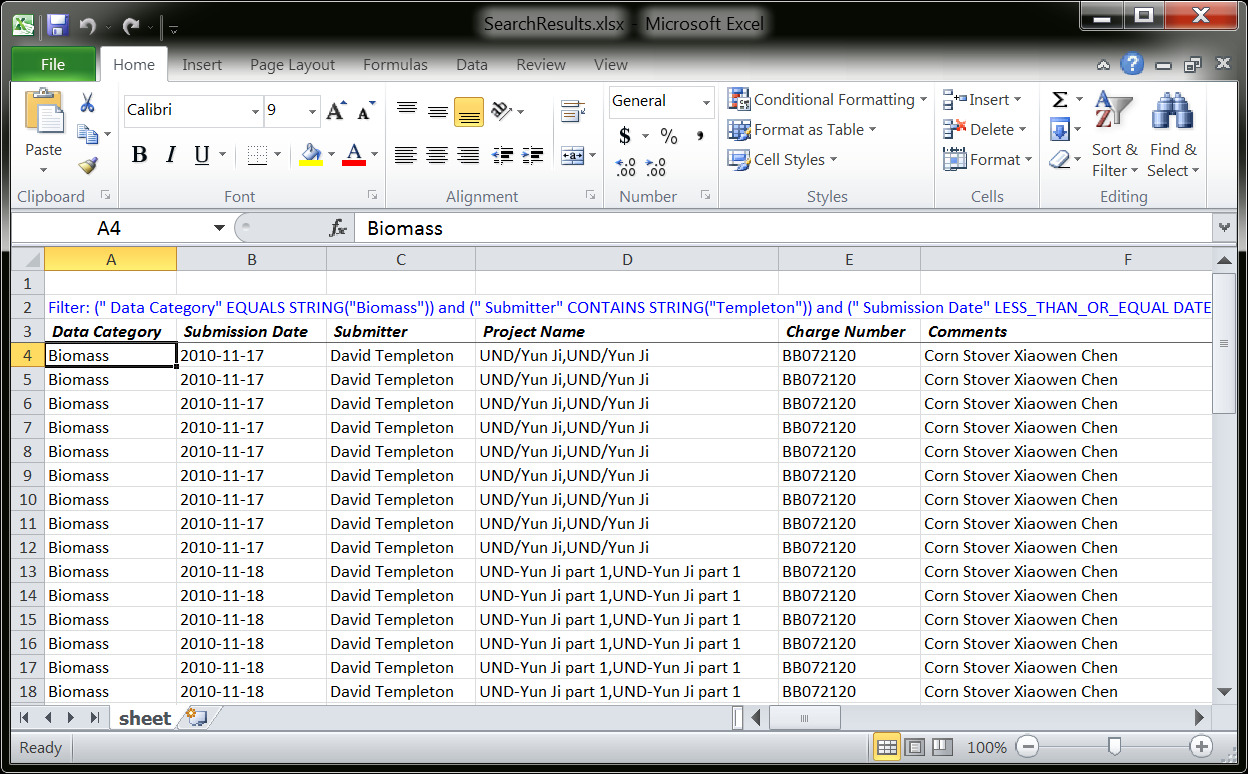


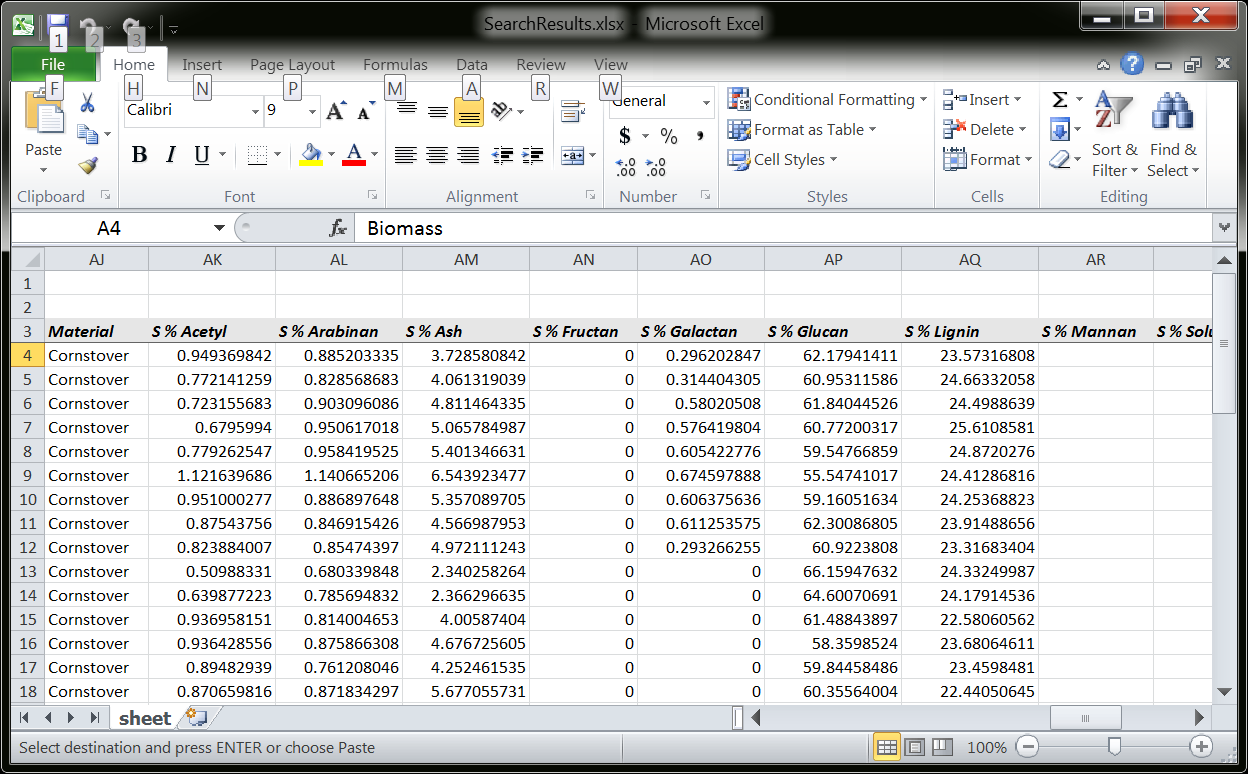
The user can conduct searches on the data that has been uploaded to the app. The fields that can be included in the search criteria are the metadata fields, as well as all the columns present in the data that has been uploaded for the selected Data Category. (So if you change your Data Category selection, the lists of fields in the selection list will likely change.)

Depending upon the number of matches, rendering the search results can take some time in the browser. If the progress wheel freezes, that is what’s happening; the browser has received the data and is now rendering it.

The columns included in the search results will include the metadata fields as well as any fields that were part of the search criteria.

If you want to see all the fields, click the “download search results” link in the upper right hand corner. The results will come in the form of an Excel workbook.





The search criteria will appear in blue lettering at the top of the workbook. The column headings will be color coded in that, the metadata fields will have a white background, and the background on the fields from the data source itself will be gray. The order of the data columns will be alphabetical.

These search result workbooks can take a while to assemble and download, so be patient.

The source document a row of data originated from can be downloaded by clicking the link in the “Source Document” column. If applicable, any attachments can be downloadable via a link in the “Attachments” column.

# Features accessed via a REST API

There are some features that are not accessible via the user interface, but are exposed via REST services.

## REST Endpoint: Removing a “Dataset”

### Background

In the software’s internals, a data upload is referred to as a dataset. If by chance someone wants to remove a dataset that has been uploaded, he can do so using this REST service.

### Usage Details

|  |  |
| --- | --- |
| Relative URL | /api/removeDataset/{dataset ID} |
| HTTP method | GET |

For this discussion, let’s assume the app is accessed via the following URL:

http://server.com/data-repository-app

And let’s assume the dataset you want to delete has the UUID 56956b54c95e773184ae88e7.

To delete the dataset, enter this in the browser:

http://server.com/data-repository-app/api/removeDataset/56956b54c95e773184ae88e7

Dataset UUIDs are included in search results in the “Source UUID” column.

Note: To prevent inadvertent permanent deletions, the uploaded files will not actually be deleted, but instead moved to a special directory on the server.

## REST Endpoint: Adding a new Data Category

### Background

The app’s configuration file has a setting (app.defaultSetOfDataCategories) for designating a default set of data categories. Here’s a sample of how it might look:

app.defaultSetOfDataCategories=Algae,ATP3,Biomass,NIR

When the app starts up, it assures the database is populated with these categories.

An additional way to add a category – one that doesn’t require an app restart – is via this REST call.

### Usage Details

|  |  |
| --- | --- |
| Relative URL | /api/addDataCategory?name=desired name |
| HTTP method | GET |

For this discussion, let’s assume the app is accessed via the following URL:

http://server.com/data-repository-app

And let’s assume the data category you want to add is “Spectral Analysis.”

To add the category, enter this in the browser:

http://server.com/data-repository-app/api/addDataCategory?name=Spectral Analysis

Note: Spaces are legal. Also, don’t include single or double quotes unless you want them to be part of the name.

## REST Endpoint: Repopulating the Database

### Background

One way to understand this application is that, while it does contain a database, the real source of record is comprised of the files users upload and their accompanying metadata. This is stored on the server’s file system. To facilitate retrieval of the data, the data is also ingested into a database.

If the database ever gets corrupt, or its schema ever needs changing, the app does provide a mechanism for wiping the database clean and re-ingesting the data from the data files stored on the file system.

### Usage Details

|  |  |
| --- | --- |
| Relative URL | /api/dropDatabaseAndReIngestAllDataFromOriginallyUploadedFiles |
| HTTP method | GET |

For this discussion, let’s assume the app is accessed via the following URL:

http://server.com/data-repository-app

To re-ingest the data, you’d enter this in the browser:

http://server.com/data-repository-app/api/ dropDatabaseAndReIngestAllDataFromOriginallyUploadedFiles

Note: Depending upon the quantity of data , this may take a few minutes to execute. While it is doing so, the app will still be functional. The only difference will be the data that hasn’t yet loaded will be omitted from search results.

# Developer Documentation

Please read the User Documentation before this section so you understand its context.

This app approaches persistent data in a couple of different ways. First, it stores the uploaded data in its raw form, namely files, on the server’s file system. It also stores the metadata the user submitted when he uploaded the data in a text file. Second, it extracts the data from the uploaded files and stores them in a MongoDB database. The database is used for conducting searches.

## Database

MongoDB was chosen as the database because it allows data to be stored in an unstructured manner, as JSON documents. This is beneficial because it allows us to store uploaded data without knowing ahead of time what the data’s column names will include. (Contrast this with a relational database, where field names must to be known ahead of time.)

In this app, there is no script to create the database. The Java code creates the database and its structure implicitly.

Instead of storing data in tables, Mongo stores data in the form of JSON documents in “collections.” For this app, we have four collections:

|  |  |
| --- | --- |
| Dataset | metadata regarding an upload, including what the user entered as well as where the data files are stored |
| Row | an individual row of uploaded data (most uploads will contain multiple rows) |
| Cell | an individual piece of uploaded data (analogous to the values in the cells of a spreadsheet) |
| dataCategory | metadata related to each data category, namely, what column names are been present in the data that has been uploaded for this category |
| datasetTransactionToken | tokens which indicate a dataset has begun being created, but hasn’t yet completed |

When an upload occurs, a dataset document is created, all of its rows are stored in row documents, and all of the values in the rows are stored in cell documents.

This is an example of a dataset document:

{

"\_id" : ObjectId("56969bcdc95e77229c646cf1"),

"metadata" : {

" Data Category" : "Biomass",

" Submission Date" : ISODate("2010-11-03T00:00:00Z"),

" Submitter" : "David Crocker",

" Charge Number" : "WW3G1000",

" Project Name" : "Qteros fermentation 9-23-10,Qteros fermentation 9-23-10",

" Comments" : "Corn Stover Clare Dibble",

" Source Document" : {

" originalFileName" : "dh10-20-10finalcalcs.xlsx",

" storageLocation" : "2016/01/2016-01-13\_AM-11-38-21\_751--0700/dh10-20-10finalcalcs.xlsx"

},

" Attachments" : [ ],

" Sub-Document" : "Digestion"

}

}

MongoDB automatically assigns a UUID to each document it stores (in the “\_id” element).

This is an example of a row document:

{

"\_id" : ObjectId("56969bcdc95e77229c646cf3"),

"datasetId" : ObjectId("56969bcdc95e77229c646cf1"),

"metadata" : {

" Data Category" : "Biomass",

" Submission Date" : ISODate("2010-11-03T00:00:00Z"),

" Submitter" : "David Crocker",

" Charge Number" : "WW3G1000",

" Project Name" : "Qteros fermentation 9-23-10,Qteros fermentation 9-23-10",

" Comments" : "Corn Stover Clare Dibble",

" Source Document" : {

" originalFileName" : "dh10-20-10finalcalcs.xlsx",

" storageLocation" : "2016/01/2016-01-13\_AM-11-38-21\_751--0700/dh10-20-10finalcalcs.xlsx"

},

" Attachments" : [ ],

" Sub-Document" : "Digestion"

},

"data" : {

"\_Sp\_Row" : 3,

"Sample\_Sp\_ID" : "P20100923 Qteros 1a",

"Tracking\_Sp\_ID" : 2711,

"Analyst" : "Deb Hyman",

"Material" : "pretreated corn stover",

"TRB\_Sp\_Number" : "3842-44",

"Date\_Sp\_Hydrolyzed" : ISODate("2010-10-19T06:00:00Z"),

"Lab" : "AFUF 203",

"Auto\_Sp\_Clave" : "Dishwash",

"LC\_Sp\_Acids" : "LC9",

"LC\_Sp\_Carbs" : "LC3",

"S\_Sp\_%\_Sp\_Ash" : 5.441253301080022,

"S\_Sp\_%\_Sp\_Whole\_Sp\_Protein" : "",

"S\_Sp\_%\_Sp\_Soluble\_Sp\_Lignin" : 1.321587665599716,

"S\_Sp\_%\_Sp\_Lignin" : 26.10890192323267,

"S\_Sp\_%\_Sp\_Glucan" : 52.136635440847755,

"S\_Sp\_%\_Sp\_Xylan" : 10.495445832679163,

"S\_Sp\_%\_Sp\_Galactan" : 0.514521500519901,

"S\_Sp\_%\_Sp\_Arabinan" : 0.8040576143593883,

"S\_Sp\_%\_Sp\_Fructan" : 0.15086282167873855,

"S\_Sp\_%\_Sp\_Mannan" : "",

"S\_Sp\_%\_Sp\_Acetyl" : 1.6348379516781815,

"S\_Sp\_%\_Sp\_Total" : 97.28651638607583

}

}

You can see that each row contains its dataset’s ID. (Basically, a foreign key, although Mongo has no join functionality.)

You might notice that the dataset metadata is also present in the row document. This is because the row collection is used for search results, and the search results include the metadata fields.

The data section contains name value pairs. The name is really the column heading, either in an uploaded spreadsheet or an uploaded CSV file.

In order to make the names either “Mongo legal,” question marks are substitured with “\_Qm\_” and decimal points with “\_Dp\_.” Spaces are substituted with “\_Sp\_” to make things unambiguous.

Mongo stores the values in such a way that it knows the value’s type, be it a string, a number, a date or a Boolean. It is because of this that user’s must specify a value’s type to conduct a search on it. (Only elements matching the type will be eligible to be considered a match.)

The metadata fields intentionally have leading spaces to assure they are uniquely named from column names in data users have uploaded.

This is an example of a cell document:

{

"\_id" : ObjectId("56969bcdc95e77229c646cf4"),

"rowId" : ObjectId("56969bcdc95e77229c646cf3"),

"columnName" : "\_Sp\_Row",

"value" : 3

}

You can see that each cell contains its row’s ID.

This is an example of a dataCategory document:

{

"\_id" : ObjectId("56969bcdc95e77229c646cef"),

"name" : "Biomass"

"columnNames" : [

"S\_Sp\_%\_Sp\_Arabinan",

"S\_Sp\_%\_Sp\_Ethanol\_Sp\_Extractives",

"S\_Sp\_%\_Sp\_Structural\_Sp\_Inorganics",

"S % Galactan",

"S % Glucan",

"L Arabinose + Potential Coeluents mg/ml",

"L HMF After mg/ml",

"L Glucose + Potential Coeluents mg/ml",

"L Lactic Acid mg/ml",

"S % Total"

],

}

As mentioned above, dataCategory documents contain the names of columns present in data that has been uploaded for the category. This list of names is re-evaluated and updated (if applicable) each time new data is uploaded. These names are used to populate the selection list containing column names on the search page.

This is an example of a datasetTransactionToken document:

{

"\_id" : ObjectId("5696b641c95e77229c68e542"),

"datasetId" : ObjectId("5696b63dc95e77229c68e541")

}

Entries in this collection are very temporary. They are used as part of a work-around for the fact that MongoDB has no atomic transaction functionality, and therefore not rollback functionality.

When data is uploaded, the first document to be created is its dataset. Once it is created, its ID is placed in the token collection. The app then creates row and cell documents. Only when all of this is complete, is the token removed.

If the app somehow dies before the process completes, upon startup, the app will see this token and remove any records related to the dataset as well as the uploaded files on the file system. It will then remove the token.

Another scenario is where data is being uploaded and an exception is thrown mid-way. In this case, exception handling code removes the data in the same manner and removes the token.

## The Document Store

Uploaded documents are stored on the server’s file system. The software deals with it all relative to a directory specified in the app’s configuration:

**app.rootDirectoryForUploadedDataFiles**=**/srv/data/data-repository-app/uploadedFiles**

This directory has three subdirectories:

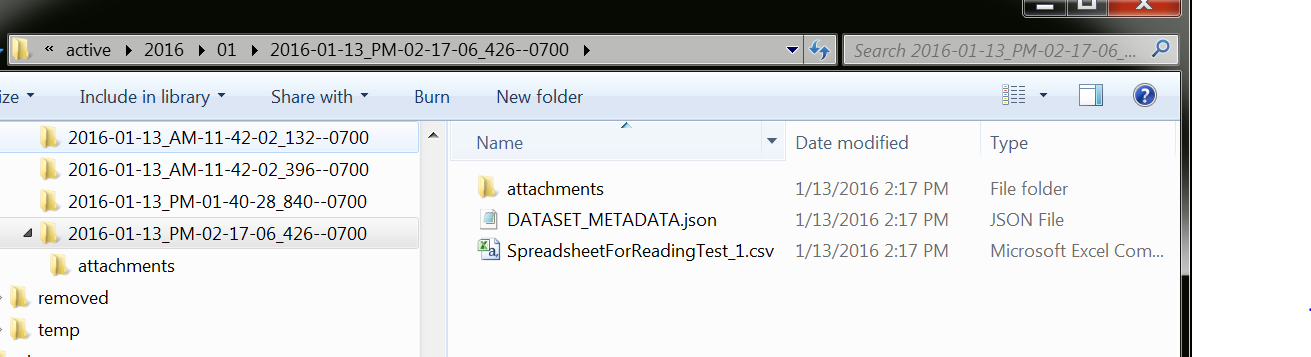
|  |  |
| --- | --- |
| active | where the uploaded files are stored |
| removed | where uploaded files are moved if the dataset has been deleted |
| temp | where files are temporarily stored in some instances |

To avoid any file name ambiguity, each upload has its files stored in its own dedicated directory within active. The file structure is such:

{year number}/{month number}/{timestamp}/

For instance:

**/srv/data/data-repository-app/uploadedFiles/2016/01/2016-01-13\_PM-02-17-06\_426--0700**



Any files uploaded as attachments will be placed in the “attachments” directory.

The DATASET\_METADATA.json file contains the same content as the document in the dataset collection.

## Java Code

The app is organized with the following tiers:

|  |  |
| --- | --- |
| **Architectural Layers** | **Implementing Technologies** |
| User Interface (UI) | HTML5, CSS3, Javascript, Angular, Bootstrap |
| REST Services | Java (Spring REST Boot) |
| Business Objects (BO) | Java |
| Data Access Objects (DAO) | Java (MongoDB’s “Java driver,” which is a Java library) |
| Data storage | MongoDB, server’s file system |

### Source Code Organization

The Java code is in src/main/java

The application’s root package is gov.energy.nrel.dataRepositoryApp.

Within this package is a class called DataRepositoryApplication. This class is significant in that it initializes the application. The current implementation does the following:

* initializes business objects, which includes, if necessary, creating the database, its collections and each collection’s indexes
* assures the default data category names (designated in the configuration file with the app.defaultSetOfDataCategories setting) are present in the database
* attempts to clean up any left-over data from previously incomplete data uploads

Note: The other classes in the root package can be disregarded. They are only applicable if you build and deploy the app as a Spring Boot executable (which means it contains a Tomcat container as well). This was developed to be deployed as a war file, so all bets are off regarding the state of these files. They have been left just in case they might be helpful as a starting point if someone wants to pursue that direction in the future.

Within that are the following packages:

|  |  |
| --- | --- |
| restEndpoints | classes related to the REST endpoints, marshalling and unmarshalling, but not real work |
| bo | classes that “do the work” |
| dao | classes that interface with storage, be it the database or the file system |
| model | classes that embody data |
| settings | the class embodying the application settings |
| utilities | utility classes, notably, the code for parsing the uploaded data files |

Some thoughts:

The model objects generally wrap Mongo database objects. Part of the purpose of this was to encapsulate the marshalling and unmarshalling of data related to the database layer, but also to make the business object layer DAO implementation agnostic. Whether that was actually accomplished is unknown; the implementation got a little messy.

An instance of the DataRepositoryApplication is instantiated by Spring.

The DataRepositoryApplication creates the business object inventory upon initialization and makes it available during the life of the application.

Each business object has the DataRepositoryApplication injected into it upon initialization, making it so each business object can access any other business object via the DataRepositoryApplication.

The DataRepositoryApplication is injected into all of the REST endpoint classes, making it and the business objects globally available to them.

Each business object is written to a particular DAO implementation.

#### Unit Tests

The business object and data access layers have fairly good unit test coverage. They actually exercise the database by creating a new database specifically for testing (a very cool capability MongoDB provides) and dropping it after each test. Although this seems like it wouldn’t be very performant, it actually is!

#### IDE: IntelliJ IDEA

The app was developed using IntelliJ IDEA. The project file is included in source control.

#### BO and DAO Layer Organization and Its Multiple Design Approaches

The organization of the BO and DAO layer looks a little weird. That’s because the app went through a progression of three different approaches to the design of the data in Mongo.

One approach involved no cell collection at all, only rows. That one was abandoned because I couldn’t determine a way to may it performant on the searches, as indexes appeared to be impossible. But that approach seemed like it would have been the most straightforward with the least duplication of data.

Another abandoned approach involved creating new cell collections on the fly, one for each data column name. Searches were broken down into separate searches for each filter criteria and conducted on the appropriate column collections. The search performance was rapid since the collections were relatively small. But data uploads took a long time. Also, it was a little unconventional to have thousands of collections. But this approach was embarked upon because, initially, it didn’t look like the next approach was adequately performant. (I *believe* this was eventually addressed with proper collection indexing.)

The approach that proved to be the best involved a single cell collection.

I have left the code for all three approached intact in case someone ever wants to revisit them. In particular, the approach involving no cell collection might prove fruitful. But it seemed to involve advanced Mongo skills to approach the queries. (Note: It may be that the abandoned approaches no longer work correctly, as the application evolved a lot after they were abandoned.)

|  |  |  |
| --- | --- | --- |
| **Design Approach** | **Java Package** | **Prefix on**  **class names** |
| only rows | abandonedApproaches\everythingInTheRowCollectionApproach | r\_ |
| mulitple cell collections | abandonedApproaches\multipleCellCollectionsApproach | m\_ |
| one cell collection | singleCellCollectionApproach | s\_ |

The code the designates which approach is “wired up” into the application is in the DataRepositoryApplication.initializeBusinessObjects() method, which, as it ended up, instantiates a s\_BusinessObjectsInventory object.

#### How Searches are Conducted

Searches are conducted in a few steps:

1. For each search criteria, the app counts the number of matching documents from the cell collection.
2. It then takes the criterion with the lowest count and executes it, retrieving row IDs from the cells. These represent a superset of rows that will eventually constitute the result set.
3. It takes the next lowest count criterion and executes it, but this time adds an “in” clause, limiting the results to only cells from rows that were matches in step 2.
4. Step 3 is repeated until either there are no more matches or the criteria have been exhausted.
5. Armed with the list of matching row IDs, the app performs a straight query against the row collection, pulling the rows in question for display to the user.

Because the cell collection is indexed on columnName and value, these queries are quick.

#### Spreadsheet Parsing

The Excel workbooks are parsed using an open source library called POI (<https://poi.apache.org/>). It seems to work fine in a lot of cases, but we’ve had some very large spreadsheets that caused it to shoot its memory usage through the roof and cause a garbage collection exception. I’ve observed it taking up multiple gigs for a 38 megabyte workbook! It seems to me there must be something wrong with the library. I searched the web for solutions and, though I found none, I found many others who have encountered the same thing.

The work-around on this is to have users export the data they want the app to ingest into a CSV file and upload *that*, including the original workbook as an attachment.

### User Interface Code

The UI code is contained in src/main/resources/static.

The HTML files are:

index.html

pages/uploadData.html

pages/findData.html

They use AngularJS as well as Boostrap.

The CSS file unique to the app is:

css/app.css

The javascript file unique to the app is:

js/app.js

External javascript and CSS files are contained in:

bower\_components

It is called bower\_components because a tool called Bower was used to pull them (http://bower.io), and it places them there by default. A file called bower.json is the script for pulling them (refer to Bower’s documentation for usage), though they shouldn’t need to be re-pulled because the files in the bower\_components directory have been added to source control. (This is so that the app’s dependencies are assured to always be present.)

There is not a lot to report regarding the UI code. It’s pretty simple, but you won’t understand it if you don’t understand AngularJS. Also, the controls for uploading files were a major pain the butt because Angular does not integrate with them. I had to do a lot of research to find a work-around.

## Application Configuration

The configuration of the application is in two files: one that contains default settings and one that can be put on the classpath to override those settings for a particular environment.

The default file is:

src/main/resources/data-repository-app\_\_defaults.properties

The override files are:

toBeCopiedToTomcatLibDirectory/local/data-repository-app\_\_envSpecificOverrides.properties

toBeCopiedToTomcatLibDirectory/dev/data-repository-app\_\_envSpecificOverrides.properties

toBeCopiedToTomcatLibDirectory/test/data-repository-app\_\_envSpecificOverrides.properties

toBeCopiedToTomcatLibDirectory/prod/data-repository-app\_\_envSpecificOverrides.properties

It’s not a good practice to keep environment specific config files in source control. But, for now, they don’t contain any sensitive information, so they’re there for convenience.

The app’s configuration file has a setting (app.defaultSetOfDataCategories) for designating a default set of data categories. Here’s a sample of how it might look:

app.defaultSetOfDataCategories=Algae,ATP3,Biomass,NIR

When the app starts up, it assures the system is populated with these categories.

### Multipart File Settings

There are settings appended with the word “multipart.” These are related to the site allowing users to upload files. The thing to note is that these settings set a cap on how large the uploaded files can be. I had to increase it a couple of times in order to upload some of the larger real-world files I was given to test with.

# Security

There were no security requirements for this app, as the app is targeted for internal use, so there are no security features.

# Admin Documentation

## Installing MongoDB

Go to <https://www.mongodb.org> and download MongoDB.

Install it.

## Starting MongoDB

At the time of this development, MongoDB came with two storage engines. One called “WiredTiger” was represented as being more performant. Therefore, I recommend we use it.

To accommodate that, as a one-time setup task, create a directory somewhere on the machine that WiredTiger can use to store data. For instance, c:/mongodb\_wiretiger.

If you specify its use at startup, Mongo will use WiredTiger. For convenience sake, you might create a script to launch Mondo using WiredTiger. Its contents would follow this form:

*mongod.exe -dbpath <path to storage directory> -storageEngine wiredTiger*

Using the directory example above, it would look like this:

*mongod.exe -dbpath c:/mongodb\_wiretiger -storageEngine wiredTiger*

Then, whenever you wanted to start MongoDB, you’d use that script. (Executing the command at a command prompt would work just as well.)

## Stopping MongoDB

When MongoDB shuts down, it does cleanup tasks. Without these tasks, data will be put in an unstable state and the next time Mongo starts, will begin by doing a (sometime time-consuming) repair operation. You may have to manually kick the repair operation with:

*mongod.exe –repair*

To allow Mongo to shut down gracefully, doing the following:

1. Launch Mongo’s command line tool:

*mongo.exe*

1. Enter the following commands:

*use admin*

*db.shutdownServer()*

## Mongo Best Practices

I would recommend looking at resources like this for best practices for deploying MongoDB:

<https://www.google.com/search?q=mongodb+best+practices>

That being said, I didn’t pick this database for its big data nor high availability features. Neither of those are hard requirements for this project. So I don’t anticipate the fact that we have no server replication planned will be any sort of a problem.

This is the contact I’ve interfaced with at MongoDB. I’ve had phone calls with him and exchanged e-mails. They seem to place special emphasis on getting government agencies to adopt Mongo in their projects.

James Kerr

Director, Solutions Architecture

phone: "646.623.6460",

email: [james.kerr@mongodb.com](mailto:james.kerr@mongodb.com)

location: Washington, D.C.

## Memory Settings

During testing we encountered a problem with memory usage that’s an inherent problem with the library being used to parse Excel files. Unfortunately, I’m not aware of any other library that can be used. It would be smart to have a relatively large chunk of heap memory dedicated to the app for Excel files that may be uploaded.

Also, we’ll want a bit of RAM for the MongoDB as well. It is designed to have all of a database’s contents in memory at all times.