Data-Repository-App Documentation

Prepared by Mike Brown

mike.public@superbrown.com

January 2016

[Introduction 4](#_Toc447147043)

[User Documentation 5](#_Toc447147044)

[Upload Data screen 5](#_Toc447147045)

[Data Categories 5](#_Toc447147046)

[Source Document 6](#_Toc447147047)

[Attachments 6](#_Toc447147048)

[Large Excel Files 6](#_Toc447147049)

[Find Data screen 7](#_Toc447147050)

[Features accessed via a REST API 10](#_Toc447147051)

[REST Endpoint: Removing a “Dataset” 10](#_Toc447147052)

[Background 10](#_Toc447147053)

[Usage Details 10](#_Toc447147054)

[REST Endpoint: Adding a new Data Category 10](#_Toc447147055)

[Background 10](#_Toc447147056)

[Usage Details 10](#_Toc447147057)

[REST Endpoint: Repopulating the Database 11](#_Toc447147058)

[Background 11](#_Toc447147059)

[Usage Details 11](#_Toc447147060)

[Developer Documentation 12](#_Toc447147061)

[Database 12](#_Toc447147062)

[dataset 13](#_Toc447147063)

[row 13](#_Toc447147064)

[cell 14](#_Toc447147065)

[dataCategory 14](#_Toc447147066)

[datasetTransactionToken 15](#_Toc447147067)

[The Document Store 15](#_Toc447147068)

[Code 16](#_Toc447147069)

[Introduction 16](#_Toc447147070)

[Source Code Organization 16](#_Toc447147071)

[Architectural Elements 17](#_Toc447147072)

[BO and DAO Layer Organization and Its Multiple Design Approaches 17](#_Toc447147073)

[How Searches are Conducted 18](#_Toc447147074)

[Spreadsheet Parsing 18](#_Toc447147075)

[Legacy Data Ingestion 19](#_Toc447147076)

[Unit Tests 19](#_Toc447147077)

[User Interface Code 19](#_Toc447147078)

[Application Configuration 20](#_Toc447147079)

[Multipart File Settings 20](#_Toc447147080)

[Application Deployment 20](#_Toc447147081)

[Miscellaneous 21](#_Toc447147082)

[IDE: IntelliJ IDEA 21](#_Toc447147083)

[Maven 21](#_Toc447147084)

[Tomcat Version 21](#_Toc447147085)

[Source Code 21](#_Toc447147086)

[Jenkins Jobs 21](#_Toc447147087)

[The Servers 21](#_Toc447147088)

[Security 22](#_Toc447147089)

[User Authentication and Authorization 22](#_Toc447147090)

[Cross-Site Scripting Protection 22](#_Toc447147091)

[Admin Documentation 23](#_Toc447147092)

[Application Deployment 23](#_Toc447147093)

[Installing MongoDB 23](#_Toc447147094)

[Starting MongoDB 23](#_Toc447147095)

[Stopping MongoDB 23](#_Toc447147096)

[Mongo Best Practices 24](#_Toc447147097)

[Memory Settings 24](#_Toc447147098)

[Multipart File Settings 24](#_Toc447147099)

[Tomcat 24](#_Toc447147100)

[Application URL 24](#_Toc447147101)

[Tomcat Version 24](#_Toc447147102)

[Deploying Multiple Instances Of The App On The Same Server 25](#_Toc447147103)

[The Servers 25](#_Toc447147104)

[Jenkins Jobs 25](#_Toc447147105)

# 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.

The user interface is browser based and REST based.

Users can upload data in the form of Excel workbooks or comma-separated-value (CSV) files.

Any kind of file(s) can also be uploaded as an “attachment.”

The uploaded data doesn’t need to conform to any particular format other than the first row containing a non-blank left cell will be interpreted as containing column names, and everything below it will be considered the values associated with those column names.

Searches are in the vein of, “Show me all rows where the value in the age column is a number equal to or greater than 18.” “Show me all rows where the value in the name column contains the string ‘Mike’.” Any rows not containing a column with the name of interest will, by definition, not be included in the search results.

Multiple search criteria can be used, and they are executed as a logical AND.

The raw data is stored in a database.

The uploaded files are stored on the server’s file system for reference.

The application provides a mechanism for searching for data rows using user-defined criteria.

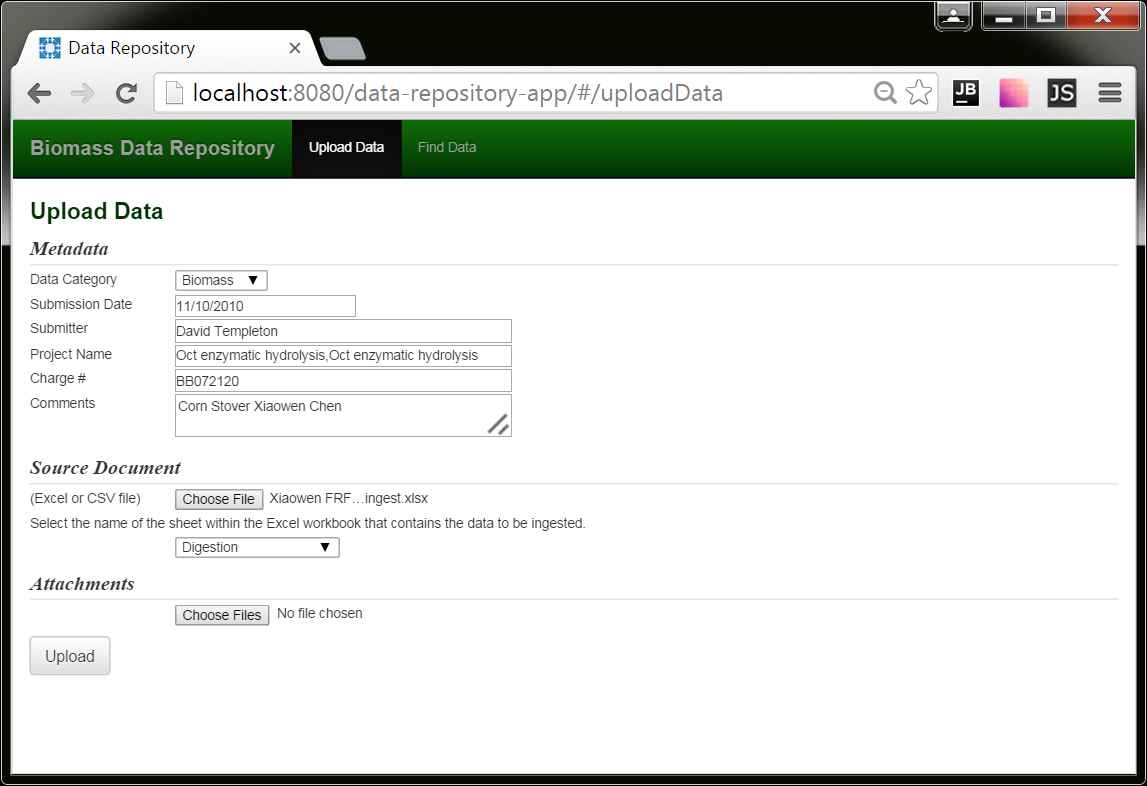
Search results include links to the originally uploaded files and any attachments.

The application was designed and developed by Mike Brown (mike@superbrown.com) for the three month period from October through December 2015.

# User Documentation

The user interface consists of two screens, one for uploading data and one for 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 name
* charge number
* comments

### Data Categories

When data is uploaded or searched upon, it is done so within the context of a chosen “data category.” Data categories segregate data of different types from one another. Essentially, each data category is a separate logical data store.

### Source Document

The uploaded source document is the document containing the data to be ingested by the application. The application can ingest Excel workbooks or CSV files. If the source document the user selects is an Excel workbook, a selection list will appear containing the list of sheets contained within that workbook. The user will use this list to indicate which sheet contains the data he wants ingested. Source documents are referenced in search results and can be downloaded from the search screen (covered later).

### Attachments

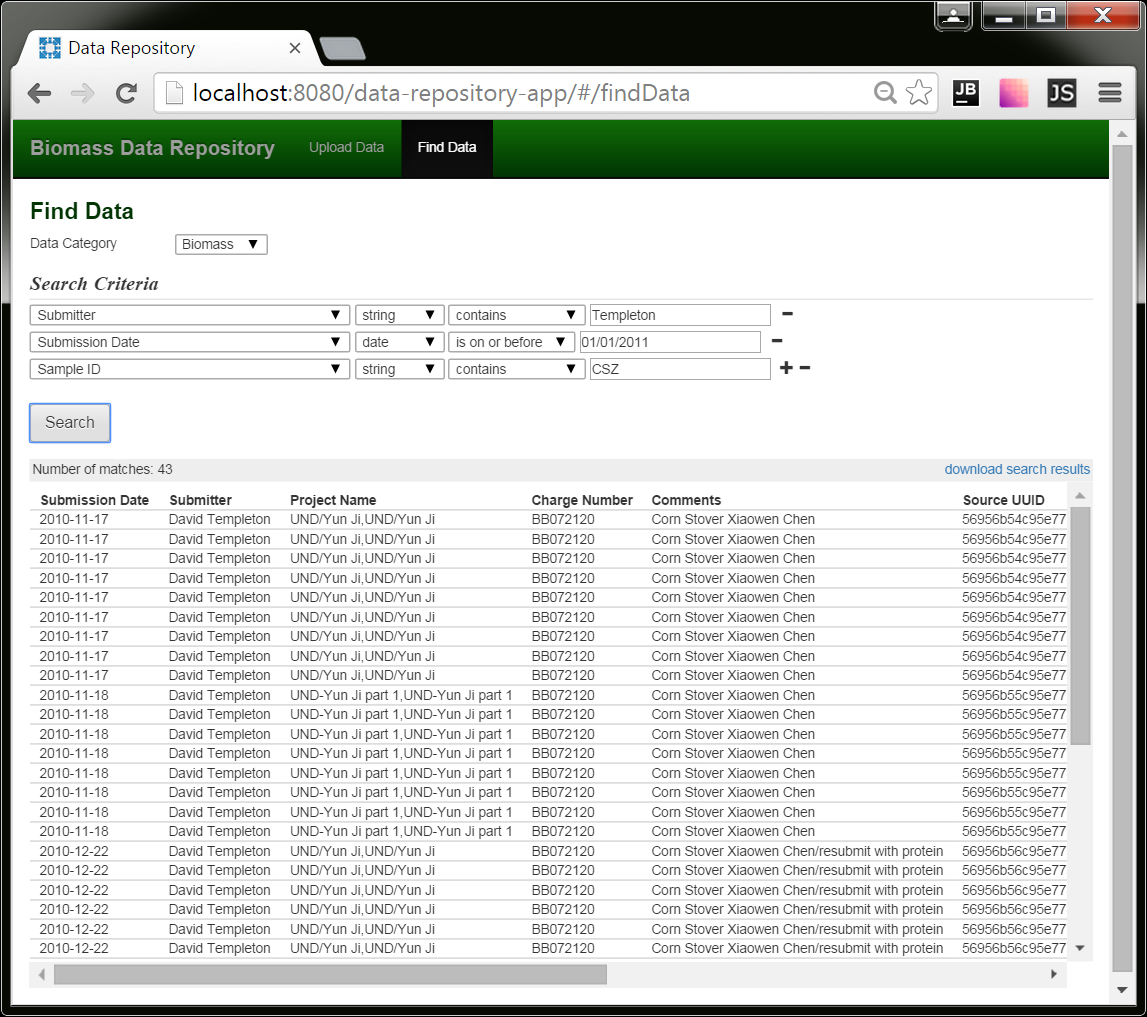
Users can also include files they consider “attachments.” Attachments can be any type of file. Like source documents, attachments are referenced in search results and can be downloaded from the search screen.

### Large Excel Files

Sometimes the application can’t parse large Excel files. The work-around is to export the data you want the application to ingest into a CSV file and upload *that*, including the original workbook as an attachment.

## Find Data screen

Users can conduct searches on data that has been uploaded. Searches can be done on metadata fields as well as column names from the raw data. Possible field names are listed in a drop down list. The names will take into account all data that has been ingested by the system for the selected Data Category.



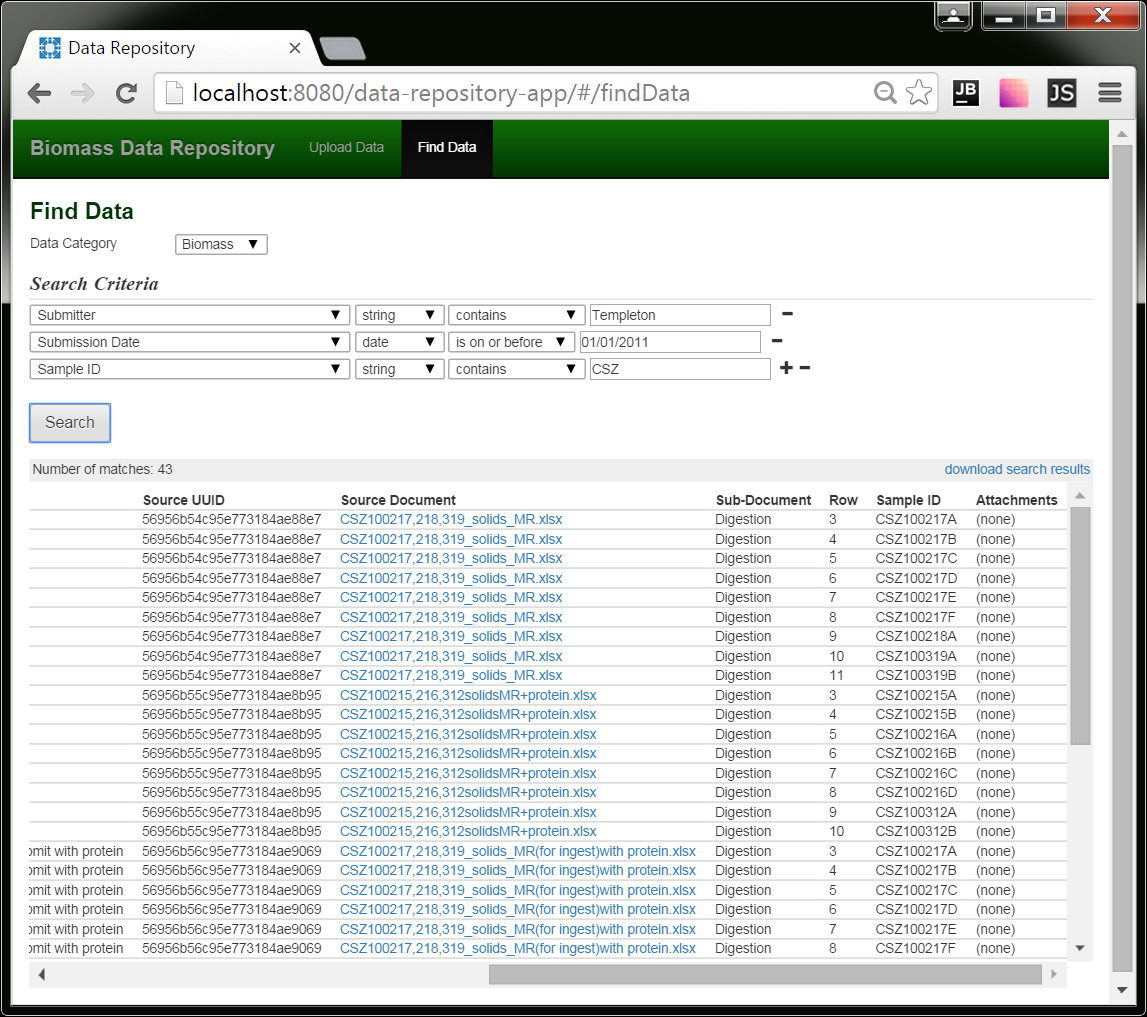
The search results are scrollable both vertically and horizontally.

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

The source document associated with any given row 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.

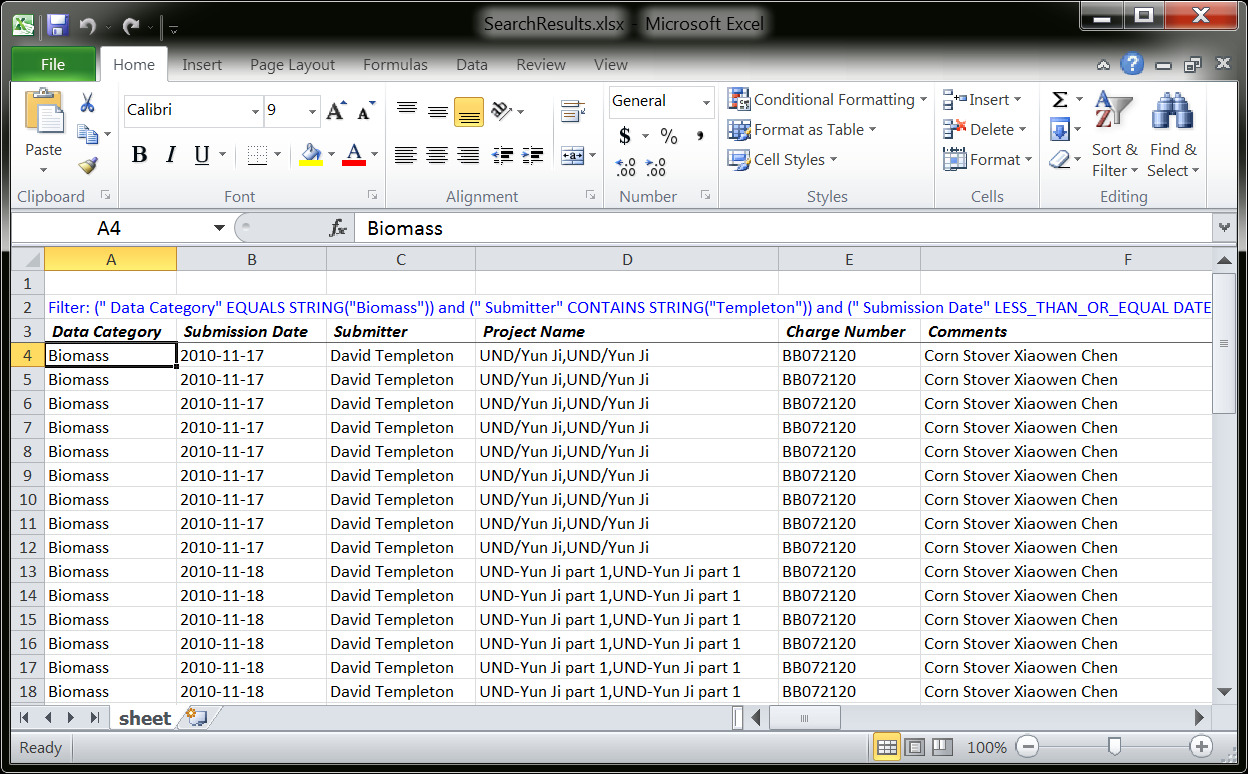
If you want to see all the fields, click the “download search results” link in the upper right hand corner, which will download an Excel workbook containing the results (see next page).

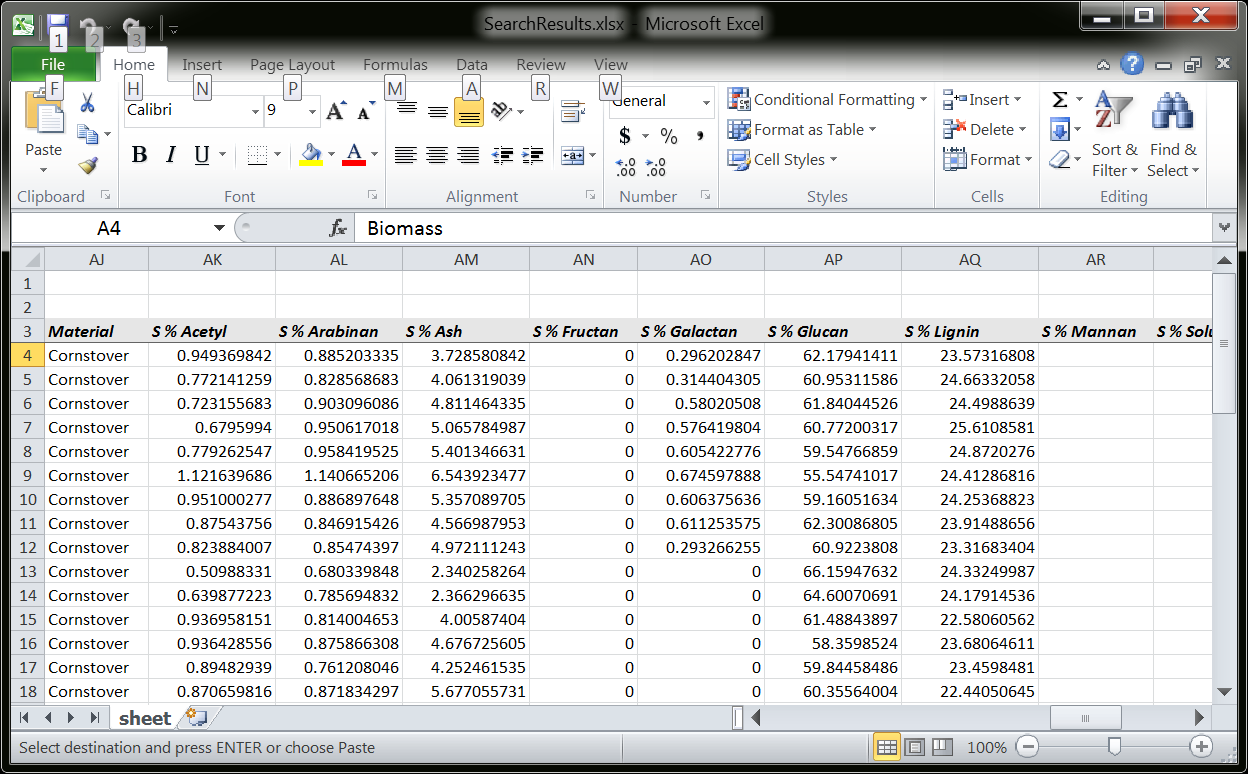
Note: Depending upon the number of matches, rendering the search results can take some time in the browser. If the progress wheel freezes, that is what is happening; the browser has received the data and is now rendering it.



The search criteria will appear in blue lettering at the top of the workbook. The column headings will be color coded in that, metadata fields will have a white background, data fields, gray. The data column names will be alpha-numerically sorted.

Note: Depending upon the size of the search results and the nature of their data, the workbooks can take a while to assemble and download, so please be patient.





# 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/v01/removeDataset/{dataset ID} |
| HTTP method | GET |

For this discussion, let’s assume the application 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 application’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 application starts up, it assures the database is populated with these categories.

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

### Usage Details

|  |  |
| --- | --- |
| Relative URL | /api/v01/addDataCategory?name=desired name |
| HTTP method | GET |

For this discussion, let’s assume the application 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 application 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/v01/dropDatabaseAndReIngestAllDataFromOriginallyUploadedFiles |
| HTTP method | GET |

For this discussion, let’s assume the application 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

***IMPORTANT!*** *If all the data is to be loaded without failure, it is imperative that the* app.defaultSetOfDataCategories *setting in the config file (explained later in this document) include all the data categories assigned to the data being uploaded. If an uploaded file’s data category (as specified in its DATASET\_METADATA.json file) isn’t one that’s in the database, it won’t be ingested. A good way to have an idea of what data categories you need to have in the database is to look what data categories show up in the data categories selection list on the UI. These may include ones beyond what are in the* app.defaultSetOfDataCategories *config setting because it’s possible for new categories to be added at runtime (for instance, via the REST call). It might prove beneficial at some point for this REST service’s behavior to be modified to automatically create any data categories found in the uploaded files’ metadata that aren’t included in the* app.defaultSetOfDataCategories *config setting, making the process foolproof.*

*Note: Depending upon the quantity of data , this may take a few minutes to execute. During this time the application will not be functional. Instead, the UI will display: “The system is locked due to the execution of cleanup operations. Please try again later.”*

# Developer Documentation

Readers should read the User Documentation before this section to understand its context.

This application approaches persistent data in a few different ways:

* It stores the uploaded data in its raw form, namely files, on the server’s file system.
* It stores the metadata the user submitted when he uploaded the data in a text file containing JSON.
* It extracts the data from the uploaded files and stores it 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.)

This application requires no script to create the database. If needed, the Java code creates the database and its structure upon startup.

Instead of storing data in tables, Mongo stores data in the form of JSON documents in containers called “collections.” This application creates and maintains collections:

|  |  |
| --- | --- |
| Dataset | metadata regarding each upload, including what the user entered as well as where the data files have been 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 have been present in the data that has been uploaded for each category |
| datasetTransactionToken | tokens which indicate a dataset has begun being created, but hasn’t yet completed, used to identify storage operations that need to be rolled back |

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.

### dataset

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"

}

}

Notice that MongoDB automatically assigns a UUID to each document and places in a specially designated “\_id” element.

### row

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\_Lignin" : 26.10890192323267,

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

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

}

}

You can see that each row contains its dataset’s ID. Basically, it’s a foreign key, although Mongo provides 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. In NOSQL databases, such data redundancy is common.

The data section contains name value pairs. The name is really the column heading from the original uploaded Excel or CSV file.

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

You might observe that 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. Because of this, users must specify a value’s anticipated type when creating a search criterion. Only elements matching the type will be eligible to be considered a match.

Note: The metadata field names intentionally have leading space. This is to make it unlikely their names will ever conflict with column names from data users have uploaded.

### cell

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.

### dataCategory

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. These names are re-evaluated and updated (if applicable) each time new data is uploaded. (Note, this is not done when data is removed from the system.) These names are used to populate the column names selection list on the search screen.

### datasetTransactionToken

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 a dataset. Once the dataset document is created, its ID is placed in the datasetTransactionToken collection. The application then goes on to create row and cell documents. Only when all of this is complete is the token removed.

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

Another scenario where this token will come into play is if data is being uploaded and an exception is thrown. In this case, exception handling code removes the data in the same manner and then removes the token.

## The Document Store

Uploaded documents are stored on the server’s file system. The software interfaces with it via a relative path specified in the application’s configuration file, such as this:

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

The application will create the following three directories under the root path:

|  |  |
| --- | --- |
| Active | where the uploaded files are stored |
| Removed | where uploaded files are moved when a dataset is removed |
| Temp | where the application can store files temporarily |

To avoid any file name ambiguity, each uploaded dataset will have its files stored in its own dedicated directory within active. The file structure is such:

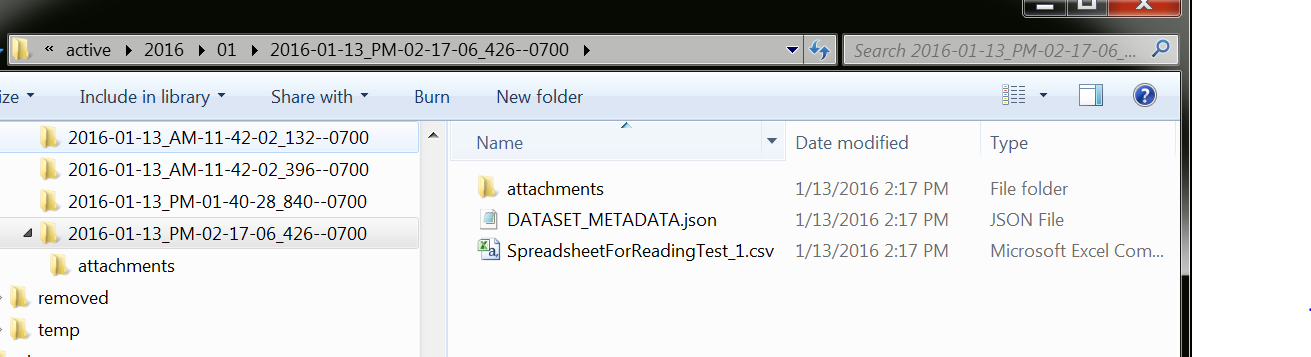
{year number}/{month number}/{timestamp}/

For instance:

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

The reason for creating a path with the year and the month is to avoid extremely bloated directories and to make it relatively easy to locate uploads based upon date. Note: The dates are the actual date of the upload, not the “submission date” the user entered on the screen.

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



The DATASET\_METADATA.json file contains all the metadata the user entered. (It’s actually the exact same content as the document in the dataset collection.)

## Code

### Introduction

The application is organized with the following logical 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 are present in the database (designated in the configuration file by the app.defaultSetOfDataCategories setting)
* attempts to clean up any left-over data from previously incomplete data uploads

Note: There are other classes in the root package, but they can be disregarded. They would only be applicable if you built and deployed the application as a Spring Boot executable (which means it would contain a Tomcat container as well). This application was developed to be deployed as a war file, so all bets are off regarding the state of these particular files. They have not been retained just in case they might be helpful as a starting point if someone wanted to pursue that direction in the future.

Within the root package are the following packages:

|  |  |
| --- | --- |
| restEndpoints | classes related to the REST endpoints, do 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 |
| utilities | utility classes, notably includes the code for parsing uploaded data files |
| settings | the class embodying the application settings |
| servletFilter | servlet filters (all of which are related to security) |

### Architectural Elements

Business objects are grouped together in a BusinessObjectsInventory object.

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, so each business object can access any other business object via the DataRepositoryApplication’s reference to the BusinessObjectsInventory object.

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

Each business object is written to a particular DAO implementation. (More on this later. There were multiple implementations developed.)

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, and also to make the BO layer DAO-implementation agnostic. (This may not have actually been accomplished; the implementation got a little messy.)

### 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 application went through a progression of three different approaches to persisting 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 appropriate column collections. The search performance was rapid since the collections were relatively small. But data uploads took a long time, probably due to the overhead of creating new collections. Also, it was probably a little unconventional to have thousands of collections (LOL). But this approach was embarked on 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 won out involved a single cell collection.

You can identify the code for each approach based on a prefix I included in their class names:

|  |  |  |
| --- | --- | --- |
| **Design Approach** | **Java Packages Have This In Their Path** | **Class Name Prefix** |
| only rows | abandonedApproaches/noCellCollectionApproach | nc\_ |
| mulitple cell collections | abandonedApproaches/multipleCellCollectionsApproach | mc\_ |
| one cell collection | singleCellCollectionApproach | sc\_ |

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, and I ran short of time to pursue it further.

(Note: It may be that the abandoned approaches no longer work correctly, as the application evolved a lot after they were abandoned.)

The code the designates which approach is “wired up” into the application is in the DataRepositoryApplication.initializeBusinessObjects() method. As the application turned out, it instantiates a sc\_BusinessObjectsInventory object.

### How Searches are Conducted

Searches are conducted in a few steps:

1. For each search criteria, the application counts the number of documents that match it in 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 search results.
3. It then 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 found to be matches in step 2.
4. Step 3 is repeated until either (1) there are no more matches or (2) the criteria have all been executed.
5. Armed with the list of matching row IDs, the application performs a straight query against the row collection, pulling the rows in question to display to the user.

Because the cell collection is indexed on columnName and value, these queries are really 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 of memory to parts a 38 megabyte workbook! It seems to me there must be something wrong with the library. I searched the web for solutions and, although I found none, I found others who have encountered the same thing.

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

### Legacy Data Ingestion

Ed Wolfrum had legacy data that he wanted ingested into the application. The code I wrote to ingest it is in the legacyDataIngest directory. It may not be needed again, but I left it there just in case.

Ed provided me a directory of spreadsheets that had been uploaded in the past into the legacy application, as well as a spreadsheet containing the names of each one and the metadata related to it. The code I wrote read the metadata and then, one by one, submitted the spreadsheets for upload by calling the REST API (just like the UI does).

The code for this is in the CodeForIngestingLegacyData class’ main() method. It takes command line arguments, and there is a comment in the code that provides an example of what those arguments might be. (When I ran it, I ran it within the IDE using an Application run/debug configuration, setting the desired command line arguments in the Program Arguments text box in the Edit Configurations dialog box.)

If you want to see what the legacy files the code was written to interface with, ask Ed Wolfrum for a copy.

### Unit Tests

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

## 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.

There is a page that documents the API (really, for developer use):

api.html

The CSS file unique to the application is:

css/app.css

The javascript file unique to the application 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 in that location 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 application’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. But explain AngularJS is beyond the capabilities of this document.

One thing I can note is that the widgets for uploading files were a major pain to implement 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 application’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 application starts up, it assures the system is populated with these categories.

### Multipart File Settings

There are some settings whose names are 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 limit 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.

## Application Deployment

For the application to work…

* Mongo needs to be installed and running.
* Tomcat needs to be installed and running.
* The application’s war has to be deployed in Tomcat’s webapps directory.
* The application’s environment specific configuration files need to be placed on the application’s classpath. (The surest way to do this is to place them in Tomcat’s lib directory.)

## Miscellaneous

### IDE: IntelliJ IDEA

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

### Maven

The build tool for this project is Maven. The script is pom.xml. The jars pulled have been copied to the inCaseSomeLibsBecomeUnavailableViaMavenSometimeInTheFuture directory for the reason embedded in its name.

### Tomcat Version

The Tomcat version used for this development was 7.0.56.

*We ran into trouble when they tried to use a different version on the DEV and TEST boxes. The problem was Tomcat wasn’t creating the temp directory necessary to save files in support of file uploads, so uploads weren’t working. I was not able to determine the root cause, but reverting to the 7.0.56 version of Tomcat made the problem go away.*

### Source Code

The source control is located in NREL’s Github:

<https://github.nrel.gov/OCIO-Data-Integrations/data-repository-app>

### Jenkins Jobs

The Jenkins jobs pull the source code from Github, build it, and then copy the war files to the appropriate servers.

DEV:

<https://apidev.nrel.gov/admin/jenkins/job/Build%20and%20deploy%20DATA-REPOSITORY-APP%20to%20DEV/>

TEST:

<https://apidev.nrel.gov/admin/jenkins/job/Build%20and%20deploy%20DATA-REPOSITORY-APP%20to%20TEST/>

### The Servers

DEV: <http://demeter:8080/nbc/>

TEST: <http://ceres:8080/nbc/>

PROD: <http://selu:8080/nbc/>

# Security

## User Authentication and Authorization

This application has no user authentication or authorization requirements, as the application is targeted for internal use, so there are no user authentication or authorization features.

## Cross-Site Scripting Protection

This app has protection in place against cross-site scripting attacks (also known as XSS attacks). Since this app is for internal use only, the likelihood that someone would try to use the app for such attacks is low. However, it did come up during security audit, so XSS protection has been incorporated into the app.

Rather than sanitizing incoming data – the approach often taken to neutralize such attacks – the application instead rejects any data containing XSS text. This data could be coming from HTML form fields or be within the uploaded Excel workbooks or CSV files. When such text is encountered, the app will display an appropriate message to the user. For instance:

“The parameter, <insert parameter name>, contains a value that could potentially be malicious.”

Or

“The file contains data that could potentially be malicious. The first encounter of such data (there may be additional examples) is located in row <insert row number>, column <insert column number>. Its "sanitized" value is: <insert value>”

This approach was taken to avoid adding data to the database from a malicious source (because it would likely be junk data). Also, it avoids the data being “silently” modified by the app without the user’s awareness. We want to avoid scenarios where the data may actually be the way the user intends, but the app nevertheless changes it because it interprets it to be potentially malicious. If the app thinks it needs to change legitimate values, the user should have visibility of that so the design of the application’s XSS detection functionality can be revisited.

# Admin Documentation

## Application Deployment

For the application to work…

* Mongo needs to be installed and running.
* Tomcat needs to be installed and running.
* The application’s war has to be deployed in Tomcat’s webapps directory.
* The application’s environment specific configuration files need to be placed on the application’s classpath. (The surest way to do this is to place them in Tomcat’s lib directory.)

(See the sections that follow for more detail on these.)

## 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 application 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.

## Multipart File Settings

There are some settings in the application’s configuration files whose names are 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 limit 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.

## Tomcat

### Application URL

As dictated by Tomcat’s deployment behavior, the application’s URL will be dictated by the war’s file name. For instance, if the file is named nbc.war, the path will be “nbc.”

### Tomcat Version

The Tomcat version used for this development was 7.0.56. We ran into trouble when they tried to use a different version on the DEV and TEST boxes. The problem was Tomcat wasn’t creating the temp directory necessary to save files in support of file uploads, so uploads weren’t working. I was not able to determine the root cause, but reverting to the 7.0.56 version of Tomcat made the problem go away.

## Deploying Multiple Instances Of The App On The Same Server

Deploying multiple instances of the app on the same server requires deploying separate wars, each within its own Tomcat container (receiving traffic on separate ports).

Background:

They have to be within separate Tomcat containers because each app will require some unique settings, to make sure their databases and uploaded file storage locations aren’t the same. The app is hard-coded to get these settings from a config file with a particular name. So if multiple apps were deployed within the same container, they’d all read the same file.

I had hoped the app could be modified to look for a particularly named config file based on the app’s path. For instance, app-01.propeties for an app instance with path app-01, and app-02.propeties for an app instance with path app\_02. While the app can indeed detect what its path is, the application framework we’re using (Spring) is wired such that the path can’t be detected until *after* app’s config file has been read. We’d need it to be the other way around.

Each app will require its config file, lib/data-repository-app\_\_envSpecificOverrides.properties, be modified in a couple of places so it doesn’t mingle its data with other apps. In particular, the following settings have to be unique:

mongoDb.databaseName

app.rootDirectoryForUploadedDataFiles

Their values are up to your discretion. Just make sure they’re unique. The app will create a database with the given name no matter what you decide to call it. And the only restriction on the root directory for the uploaded files is that the path has to actually exist.

## The Servers

DEV: <http://demeter:8080/nbc/>

TEST: <http://ceres:8080/nbc/>

PROD: <http://selu:8080/nbc/>

## Jenkins Jobs

The Jenkins jobs pull the source code from Github, build it, and then copy the war files to the appropriate servers.

DEV:

<https://apidev.nrel.gov/admin/jenkins/job/Build%20and%20deploy%20DATA-REPOSITORY-APP%20to%20DEV/>

TEST:

<https://apidev.nrel.gov/admin/jenkins/job/Build%20and%20deploy%20DATA-REPOSITORY-APP%20to%20TEST/>