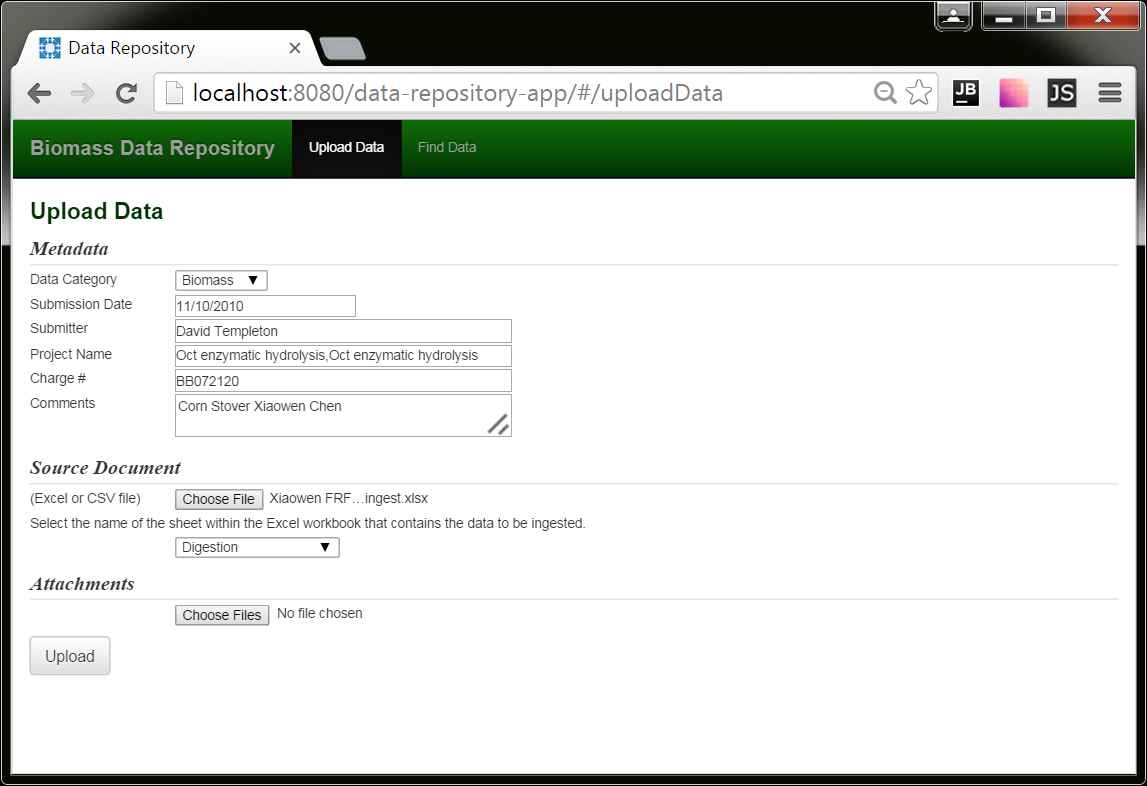
# Introduction

The Data Repository App has been developed for the purpose of storing data in grid form (i.e. rows and columns) and retrieving it later via searches. It has a browser-based UI that users can use to upload data in the form of Excel workbooks or comma-separated-value (CSV) files. The ingested data is stored in a MongoDB database. The raw uploaded files are stored on the server’s file system for reference. 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.

# 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.

### 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.

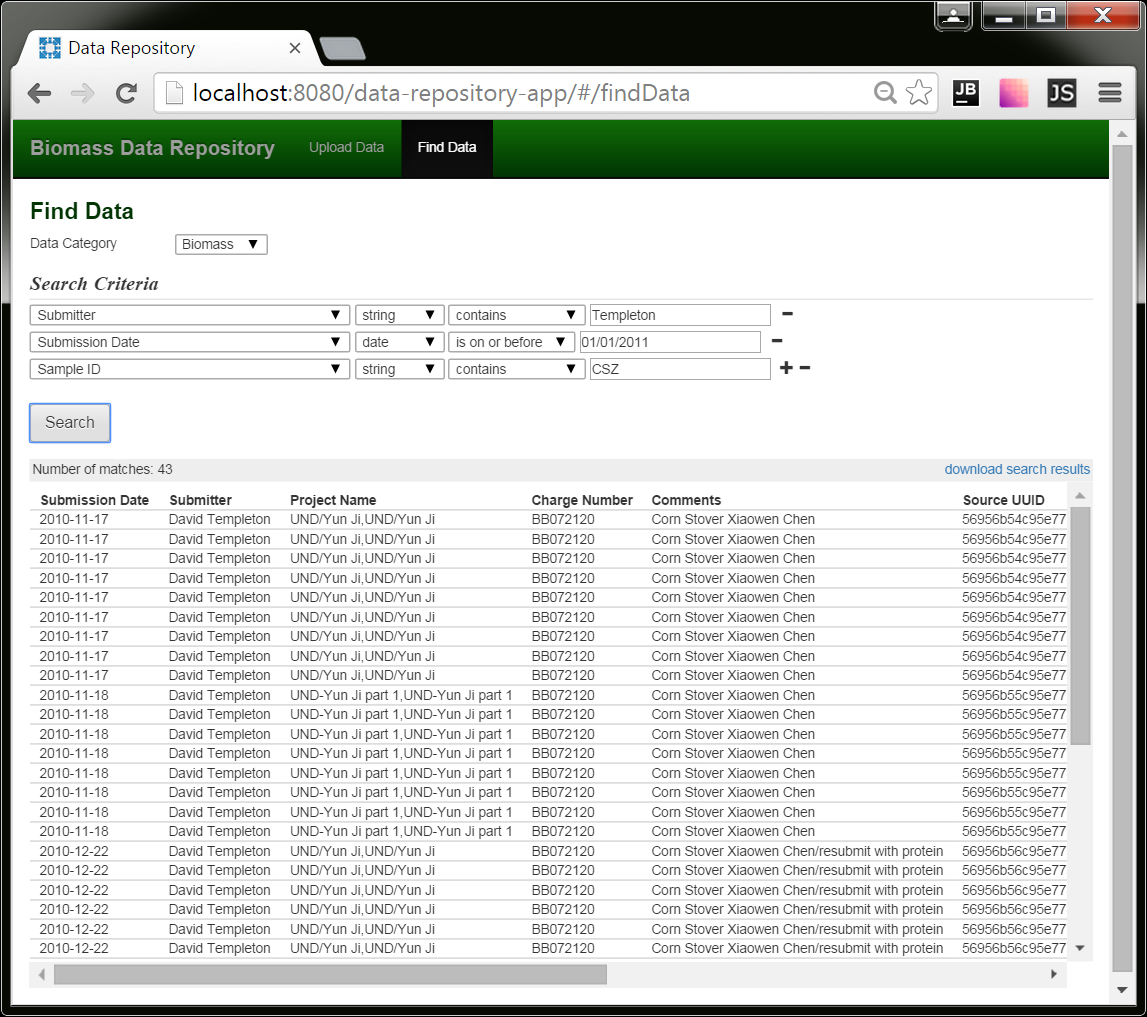
### Attachments

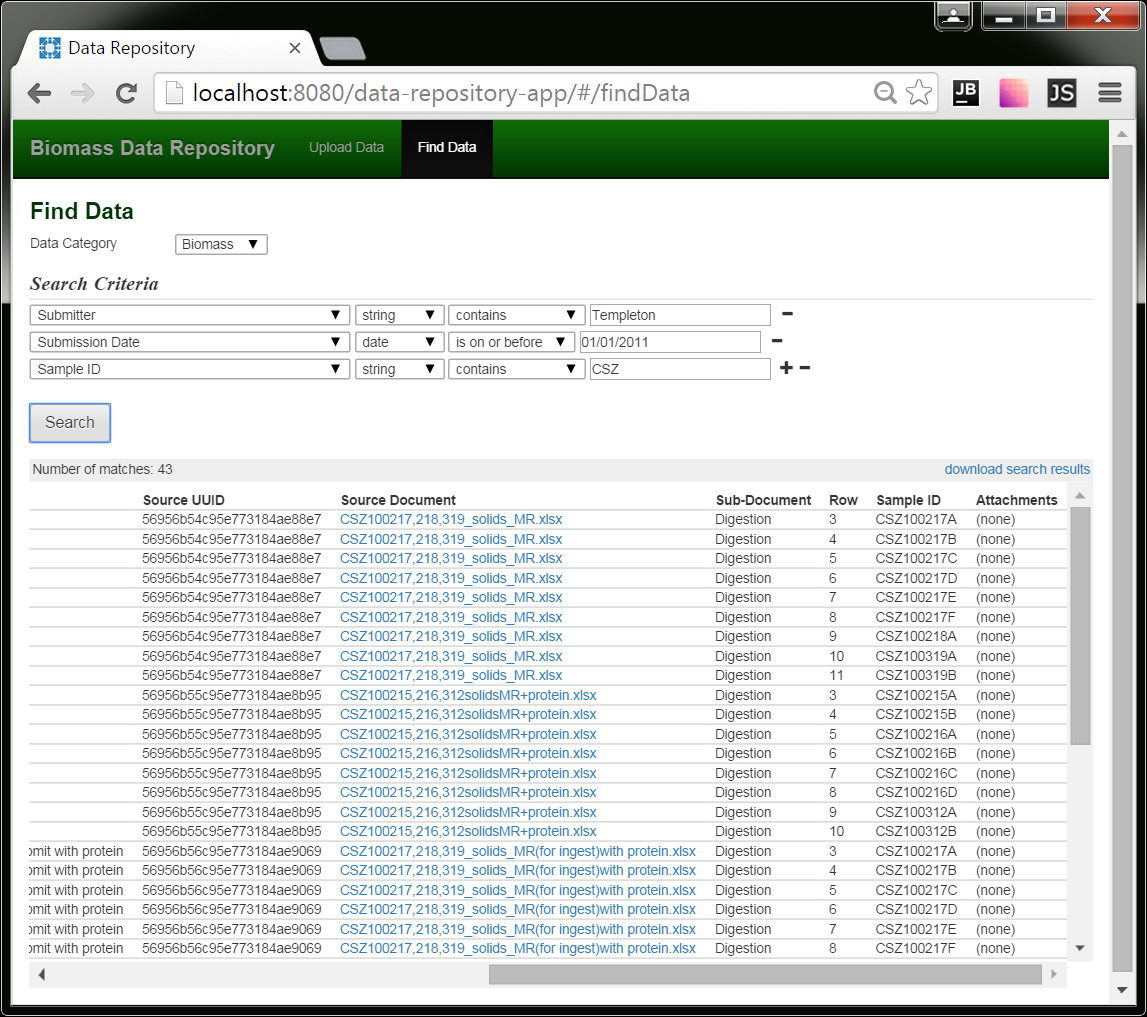
The user can also include, in the upload, files he considers “attachments.” These are retrievable later.

### 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



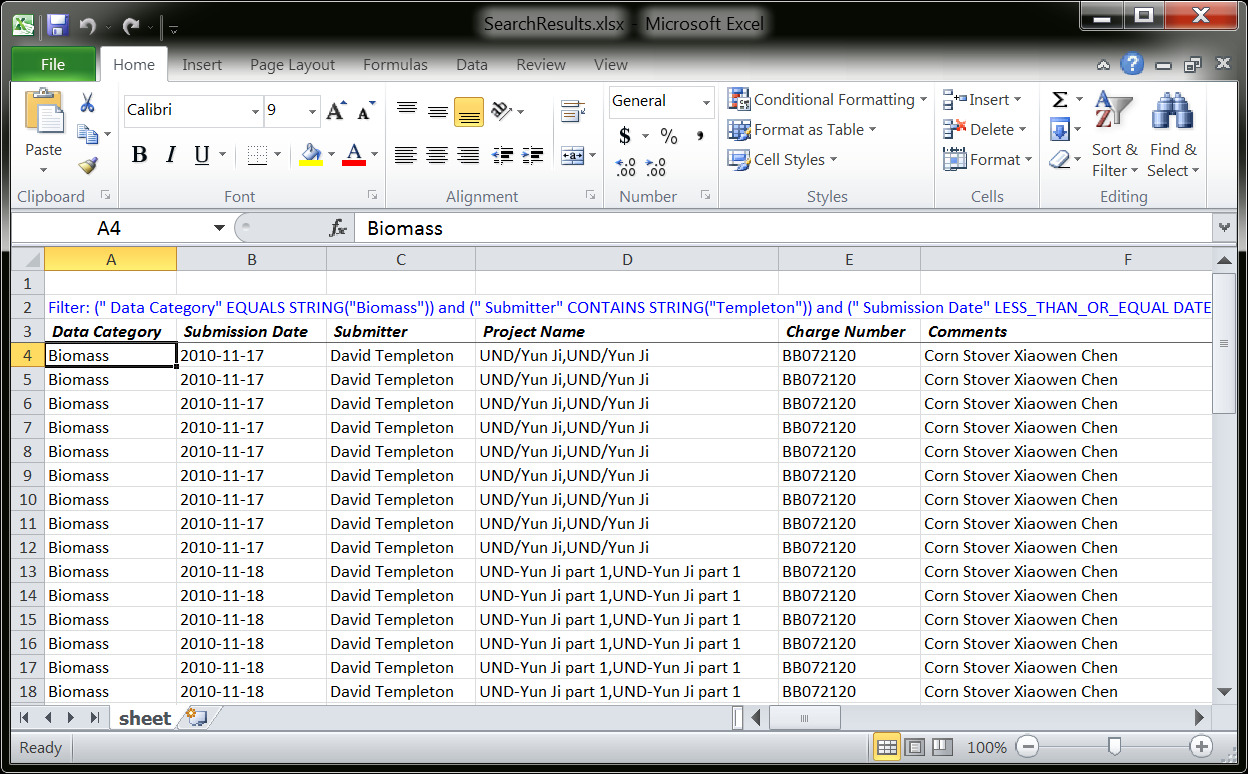


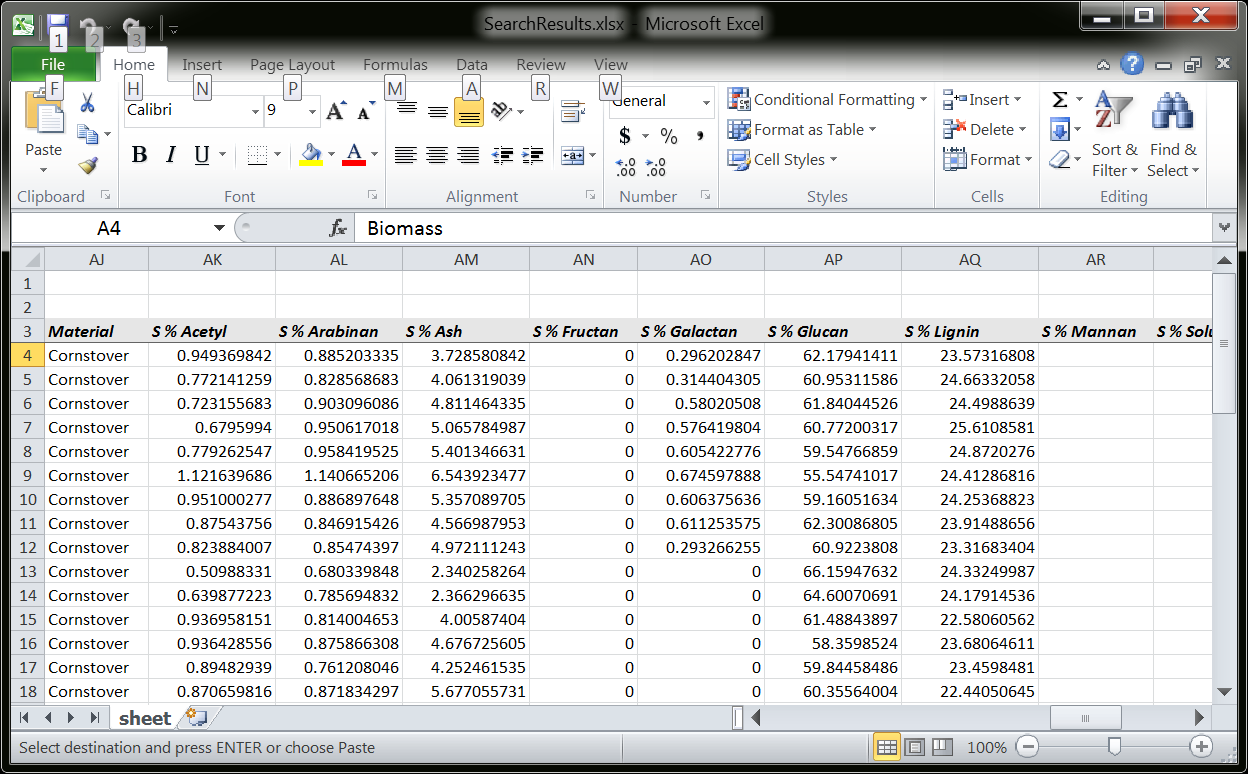
The user can conduct searches on the data that has been uploaded to the app. The fields that can be included in the search criteria are the metadata fields, as well as all the columns present in the data that has been uploaded for the selected Data Category. (So if you change your Data Category selection, 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 can be via a REST API. To call these, you’ll need to use a tool for making REST calls. An example would be the RESTClient plugin you can get for the Firefox browser (<https://addons.mozilla.org/en-US/firefox/addon/restclient>). A web search should produce any number of possible apps for making REST calls.

## Removing a “Dataset”

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.

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

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

Call this service with the HTTP *DELETE* method, using the following form:

http://myhostingserver.com/data-repository-app/api/dataset/{Source UUID}

Your REST client will allow you to select the HTTP “method” or “verb.” This will need to be DELETE (not GET or POST, or any other verb).

The Source UUID can be identified on the search results screen; it’s one of the columns returned.

To prevent inadvertent permanent deletions, the files uploaded for the dataset will not actually be removed from the server, but rather, will be moved to a special “removed” directory on the server.

## Adding a new Data Category

## Repopulating the Database

The app’s configuration files are contained in resources directory.

The app is designed in a tiered manner.

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

**Source Code Organization**

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

The HTML files are:

index.html

pages/uploadData.html

pages/findData.html

The CSS file unique to the app is:

css/app.css

The javascript 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.)

The Java code is in src/main/java

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

Within it is a class called DataRepositoryApplication. This class is significant in that it initializes the application. One of the things it does is make sure

Within that are the following packages:

app: contains the application,

bo: business objects

dao: data access objects

model:

restEndpoint:

settings:

utilities:

Applicaiton Configuration

**THE DATABASE**

The Data Repository App uses MongoDB as its data store. It was chosen because it lets you store data in an unstructured manner, as JSON documents. This was a benefit because it allowed us to store the uploaded data without knowing ahead of time what the data’s columns would include.

***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()*

**Memory**

Auto database creation

Unit tests

Different DAO approaches

File storage location

REST endpoints:

To add a new data category

To delete a dataset

To repopulate the database (using uploaded data files)