

I. Getting Started

First, it is important to be sure to have downloaded and installed all necessary components of our project's technology stack

Java 8

any Java IDE - we used Eclipse with the m2eclipse plugin for maven

Maven

Neo4j Community Edition

Apache Tomcat 7

and the project source code, of course

[Java downloads](#)

[Neo4j Download Page](#)

[Apache Tomcat 7](#)

[This guide can help set up Maven with Eclipse](#)

[Guide for installing and configuring tomcat](#)

Our OSU server address:

<http://server2.ies.cse.ohio-state.edu>

We installed Java, Neo4j, and Tomcat on the Linux server provided by Ohio State. Our system is using two ports on this server:

<http://server2.ies.cse.ohio-state.edu:8080> - The Tomcat manager app is accessed here

<http://server2.ies.cse.ohio-state.edu:8081> - The Neo4j instance is accessed here

The port that Neo4j is running on can be accessed in the file:

/home/likewise-open/IES/kuhn.615/capstone/neo4j/neo4j-community-2.1.7/conf/neo4j-server.properties

with the line:

`org.neo4j.server.webserver.port=8081`

In the /neo4j-community-2.1.7/bin folder, there are some scripts for Neo4j. You start the database by running the script "neo4j start" where start is taken as the first argument for the script. Likewise, you can run "neo4j stop" to stop the database.

Once installed and running, you can navigate to the Neo4j url at port 8081 and type commands into a console that can manipulate the database. You can clear the database and run the Database Creation Script to reformulate the database.

The project is readily imported into Eclipse. Inside the Util.java class, you may specify the location of the Neo4j instance with the line

```
public static final String DEFAULT_URL = "http://server2.ies.cse.ohio-state.edu:8081";
```

Once the project has been imported into Eclipse, use the external tools to run maven commands on the project.

[Help with using Maven with External Tools in Eclipse](#)

Create a command with the arguments: "clean install"

Each time you run this command, it will create a .war file in the path

```
/<project_root_dir>/target/
```

When in the Tomcat manager app at <http://server2.ies.cse.ohio-state.edu:8080/manager/html>, there is a box that gives you the option to upload a .war file from your computer onto the server. Simply select the .war file from the /target folder inside the project directory and click upload.

IMPORTANT NOTE:

You can rename the .war file to anything you like, but when you deploy the app to the Tomcat server, you will access the app by going to

server2.ies.cse.ohio-state.edu:8080/<war_file_name>

Thus, any functions in the file **frontend.js** that use HTTP requests to our backend's endpoints must include the <war_file_name> in its requested route. Example in frontend.js:

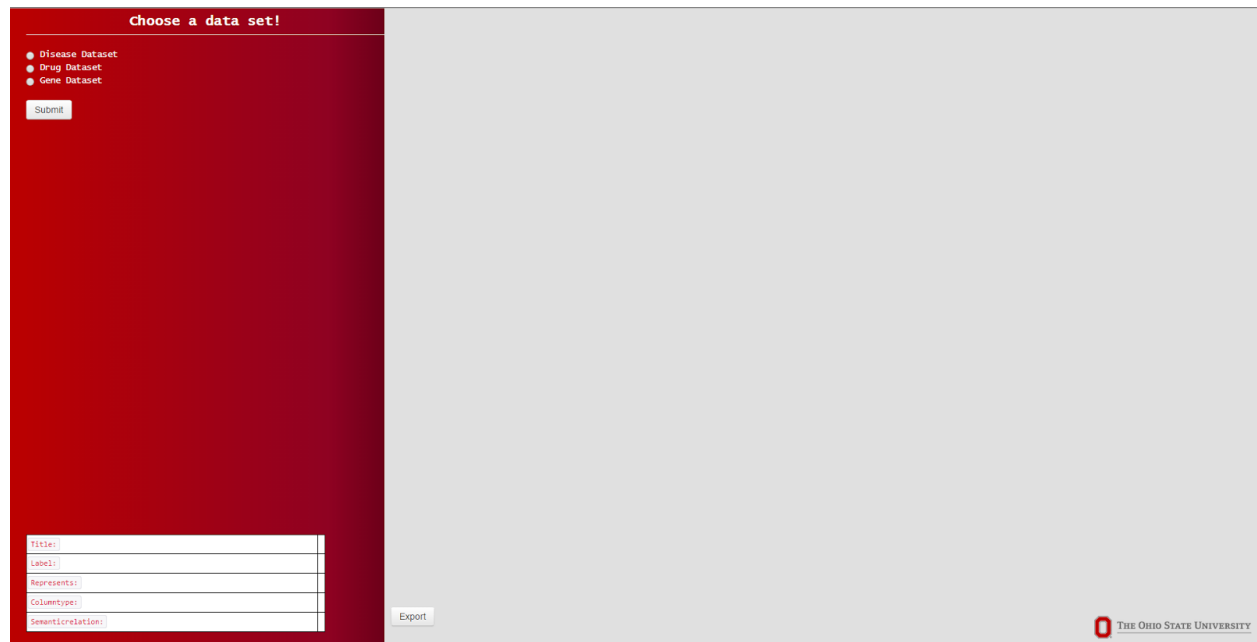
```
d3.json("/ty/datasets", function(error, data) { ... }); // line 88
```

where "ty" is <war_file_name>

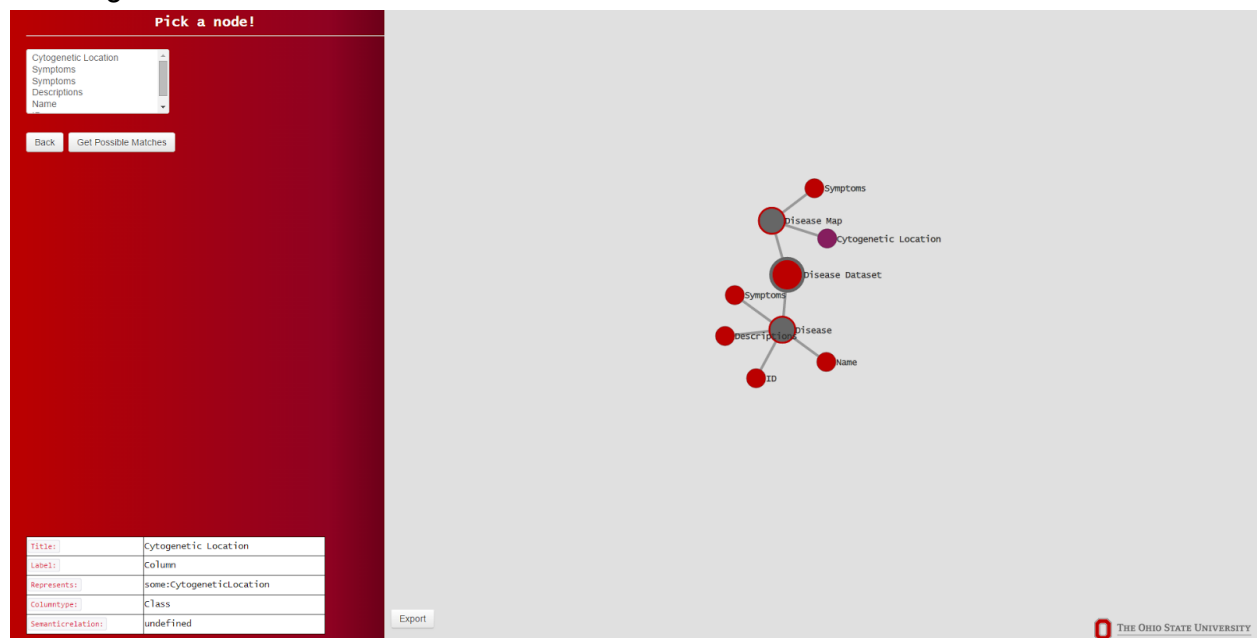
Once the app is deployed, simply click on its path created by the deployed .war file in the table and the app will load.

II. How to Use the App

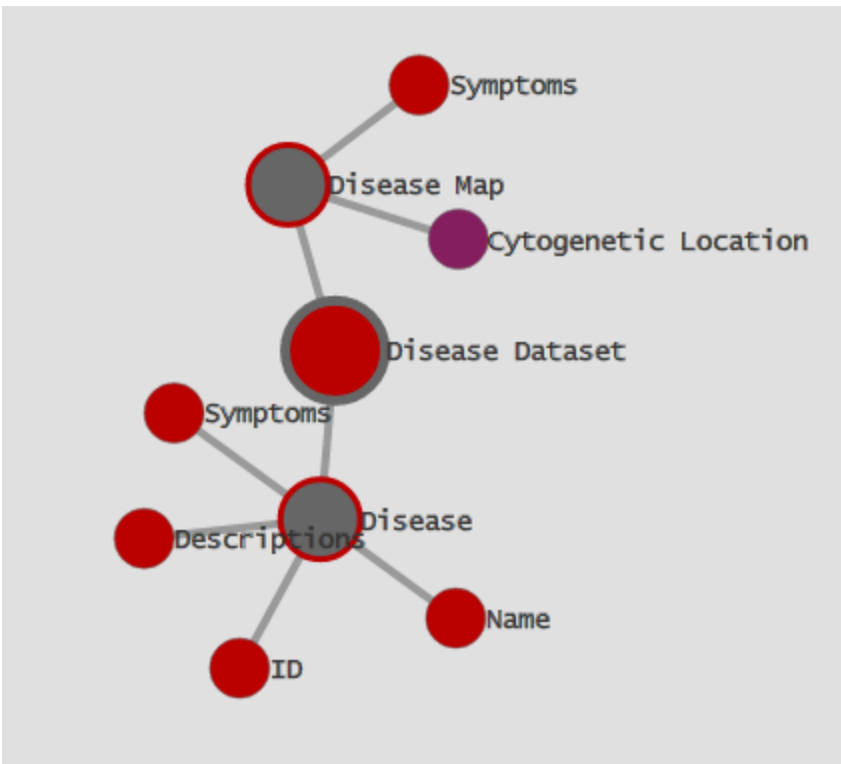
Upon starting the app, the user is presented with a view featuring a vertical console to the left and an empty space filling the rest of the screen. Inside the console, there is a list of all of the datasets present in the Neo4j instance.



Also inside the console, located at the bottom is the table display that shows meta-info about the nodes that are in the graph. After choosing a dataset to start with, the user clicks the submit button to fetch the dataset. This causes the chosen graph to be displayed in the area to the right.



The dataset appears as so:



There are 3 types of nodes:

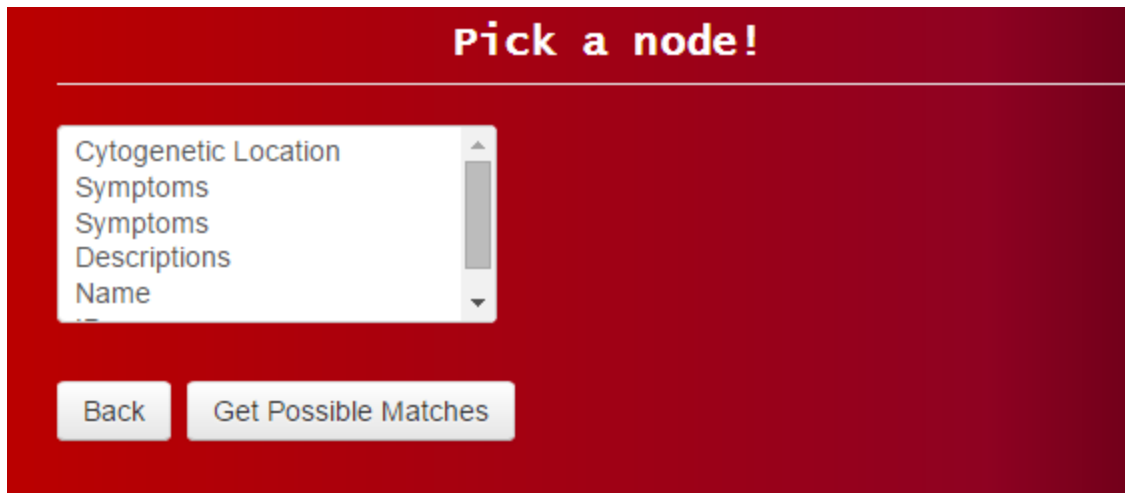
1. Dataset Nodes
2. (Join)Table Nodes
3. Column Nodes

The dataset nodes represent datasets that contain tables. These various tables are shown as the children of the dataset node. Each table is comprised of its respective column nodes. One of the above column nodes is purple in the picture, that is simply because it is currently selected, meaning the user recently clicked on it. Selecting a node will cause its metadata to populate the table in the console.

Title:	Cytogenetic Location
Label:	Column
Represents:	some:CytogeneticLocation
Column type:	Class
Semantic relation:	undefined


This metainfo table shows us the annotations of the column. You can see here that the column represents “some:CytogeneticLocation,” and it does not have a specific semantic relation.

Note now the list of datasets has been replaced by a box that allows you to choose a node in the graph and submit it for querying. You can also choose a node by clicking on it.



The screenshot shows a web interface with a dark red background. At the top, the text "Pick a node!" is displayed in a bold, white, sans-serif font. Below this text is a white rectangular box containing a list of items: "Cytogenetic Location", "Symptoms", "Symptoms", "Descriptions", and "Name". The list has a scrollbar on the right side. Below the list box are two white buttons with dark red text: "Back" and "Get Possible Matches".

Since Cytogenetic Location was selected earlier the user can now click on the “Get Possible Matches” button in order to see what queries can be ran on the column data.



The screenshot shows a web interface with a dark red background. At the top, the text "Explore different matching properties!" is displayed in a bold, white, sans-serif font. Below this text are three white buttons with dark red text: "Find Related Titles", "Find Related Entities", and "Back". The buttons are arranged vertically, with "Find Related Titles" and "Find Related Entities" stacked together, and "Back" below them.

Recall from earlier that our column *represented* “some:CytogeneticLocation,” and it did not have a semantic relation. Because of this, we see two buttons. The “Find Related Titles” button will query the database for any other columns that have the same name as the chosen column. The “Find Related Entities” button will query the database for any columns that *represent* the same thing as the chosen column. Had our selection had a semantic relation, there would be a button titled “Find Related Attributes” that would query the database for any nodes that had the same semantic relation as the chosen node. Upon clicking one of the query buttons, the graph is updated to display the results of the query.

After clicking on “Find Related Entities,” the user is presented with an updated graph showing dashed lines between the column node queried on and the new column nodes that resulted from the query. The related column entities are brought into the graph with their respective table, sibling and dataset nodes. The dashed lines indicate that the node on the opposite end resulted from the query chosen by the user.

Modify the graph!

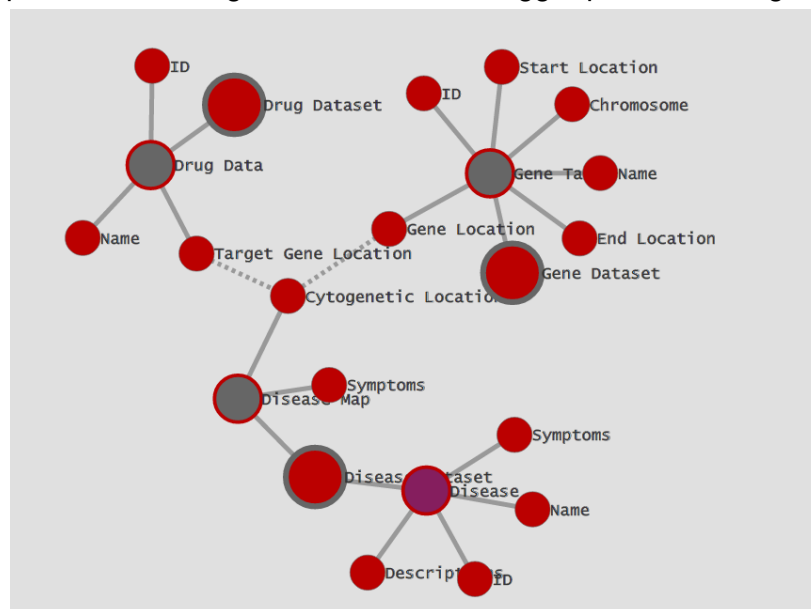
Connecting Nodes	Create Edge	Delete Node
Target Gene Location	<input type="radio"/>	<input type="radio"/>
Gene Location	<input type="radio"/>	<input type="radio"/>

[Back](#)

Title:	Target Gene Location
Label:	Column
Represents:	some:CytogeneticLocation
Column type:	Class
Semantic relation:	some:targetgene

[Export](#)

From here, the user can solidify the dashed line by saving the match. This is done by clicking on the dashed edge itself, or choosing the “Create Edge” option from the table in the console. The user also has the option to remove any of the matched nodes from the graph by selecting the “Delete Node” option which will remove the dashed edge, the matched column node, its parent and sibling nodes. Below is a bigger picture of the graph.



The next picture below shows the result of creating the edge going to “Target Gene Location” and removing the node “Gene Location” and its relatives.

Pick a node!

Cytogenetic Location
 Symptoms
 Symptoms
 Descriptions
 Name

Back
Get Possible Matches

Title:	Symptoms
Label:	Column
Represents:	mesh:Disease
Column type:	Class
Semantic relation:	undefined

Export

IMPORTANT NOTE:

The user must go all the way back to the for titled “Pick a node!” in order to initiate queries on other nodes in the current view. This form can be seen in the above picture.

Below is pictured the result of next selecting the “ID” column from the “Disease” table and matching other columns that have related attributes (AKA semantic relations).

Modify the graph!

Connecting Nodes	Create Edge	Delete Node
ID	●	●

Back

Title:	ID
Label:	Column
Represents:	undefined
Column type:	Property
Semantic relation:	dc:identifier

Export

IMPORTANT NOTE:

Only nodes that were not in the graph prior to the query get listed in the console upper left table.

This is indeed a small caveat, however, the connections can still be saved by clicking on them, turning the dashed lines into solid lines. Once the user has created his desired connections based on the data annotations, he can export the meta-info about the connections via downloading a csv file that is formatted as seen below. The user initiates this event by clicking on the “Export” button that is seen in the lower left corner of the graph display.

A	
1	Node:TargetGeneLocation_Node:CytogeneticLocation_represents:some:CytogeneticLocation_Date:4/28/2015-3:07:24AM
B	
	Node:TargetGeneLocation_Node:CytogeneticLocation_represents:some:CytogeneticLocation_Date:4/28/2015-3:26:24AM
C	
	Node:ID_Node:ID_semanticrelation:dct:identifier_Date:4/28/2015-3:34:18AM

The information contained in each csv value is as follows:

The column resulting from the query + the column node queried on + the annotation matched and its value + a timestamp of the creation

III. How the App Works Internally

Our application runs on a REST API. We have defined specific endpoints on the backend that the user can request on the front end based on input

```
// handles a get request to the route /datasets
// it is initially requested when the webpage loads in order to give the user a list of datasets to start from
get("/datasets", new Route() {
    public Object handle(Request request, Response response) {
        // limit defines a limit on the length of the response if it is not defined in the request
        int limit = request.queryParams("limit") != null ? Integer.valueOf(request.queryParams("limit")) : 100;
        // The gson.toJson simply converts the given data to JSON format for sending it in an HTTP response.
        // Here we are calling the datasets method in the service object which will query neo4j for all the
        // datasets in the database.
        return gson.toJson(service.datasets(limit));
    }
});
```

The above snippet of code from DFRoutes.java shows how to define and handle an endpoint. In this example, we define the endpoint “/datasets” and delegate a task to the service object to handle the request. The shown return statement is returning JSON-converted data resulting from the call to service.datasets(). Each endpoint will have a corresponding method within the DFService.java class. These methods will end up running cypher queries corresponding to the name of the endpoints.

Below is shown the datasets() method in the DFService class. Note that the cypher query being run here is:

Match (n:Dataset) Return n as dataset, id(n) as id

```
// request handler for initially sending all dataset nodes to front end
// limit defines the max length of the response. It is used as a parameter in requests
public Map<String, Object> datasets(int limit) {
    // limit tells the neo4j server the max length of the response
    Iterator<Map<String, Object>> result = cypher.query(
        "match (n:Dataset) return n as dataset, id(n) as id",
        map("1", limit));
    List<Map<String, Object>> datasets = new ArrayList<Map<String, Object>>();

    //result is essentially a collection of rows in a table of data returned by the query
    while (result.hasNext())
    {
        Map<String, Object> row = result.next();
        Map<String, Object> dataset = map("id", row.get("id"), "datasetNode", row.get("dataset"));
        datasets.add(dataset);
    }

    return map("datasets", datasets);
}
```

This query returns each dataset existing in the database, along with its corresponding ID. This info is used to populate the first form prompting the user to choose a dataset to start with.

The following is a list of the endpoints, their corresponding cypher queries, and how the data is used. It is common to see parameters passed in the routes to be used in the queries.

Endpoint:

/getDataset/:datasetID

(where :datasetID is a given integer param)

Query:

```
start n=node( datasetID )
match (n)-[:BELONGS_TO]-(p)-[:BELONGS_TO]-(c)
return n as dataset, labels(n)[0] as datasetType, ID(n) as datasetId, n.title as datasetName,
p.title as parentName, labels(p)[0] as parentType, ID(p) as parentId, p as parent, c.title as
childName, labels(c)[0] as childType, ID(c) as childId, c as child
```

Use:

This returns the dataset, table, and column nodes and edges of a dataset with the given ID

Endpoint:

/getTableIdForNode/:nodeID

(where :nodeID is an integer param)

Query

```
start n=node( nodeID )
match (n)-->(p)
return id(p) as tableID
```

Use:

This is used to find the parent table node ID of any given column node ID

Endpoint:

/getTable/:nodeID

(where :nodeID is an integer param)

Query:

```
start n=node( nodeID )
match (n)-->(p)
with id(p) as pid
match path=(n)-->(t)-->(d)
where id(t) = pid
unwind nodes(path) as r
return distinct r.title as name, labels(r)[0] as type, id(r) as id, r as node
```

Use:

This is used to find a given column node ID's:

1. parent table node
 2. grandparent dataset node
 3. sibling column nodes
-

Endpoint:

/matchProperty/:property/:propertyValue

Query:

match (n:Column)

where n.property = propertyValue

return n.title as name, labels(n)[0] as type, id(n) as id, n as node

Use:

This is used to find all column nodes that share a given property and property value