Getting Started

Getting a new source into DGIdb consists of two primary parts: acquiring the data itself in a usable format and then telling DGIdb what it means. The first step varies for each source so you're somewhat on your own. In lib/genome/downloaders there is an example script that scrapes the MyCancerGenome website and produces a TSV file from it. There are additional downloaders in the Genome repo under Genome/DruggableGene/Command/Import.

Once you have a a flat file with your data, it is pretty easy to get it into DGIdb. There is a declarative DSL that should allow you to simply describe the contents of your file.

There are many existing examples of this in the DGIdb repo under lib/genome/importers/<source_name>.

For each source, there are two code files that you will need to write. The first file contains a class that describes the format of the rows in your data file. The importer will use instances of this class to wrap rows of data in your file. Let's walk through an example of creating one.

Define Your Row Structure

The first class you'll write should be a subclass of Genome::Importers::DelimitedRow. This allows you to use the "attribute" method that will make up the bulk of our class. Let's say we have a data file with rows that looks something like this (assuming that our spaces are tabs for ease of reading):

Each row in this case represents an interaction between a drug and a gene. As you will have created this file, you will be familiar with its format.

```
FLT1  2321 P17948   ENSG00000102755  TIVOZANIB  KRN951|AV-951
KINASE_INHIBITOR  INHIBITOR
```

You would create a class that essentially acts as headers for your (in this case, TSV) file.

```
class MyNewSourceRow < Genome::Importers::DelimitedRow
    attribute :primary_gene_name
    attribute :entrez_gene_name
    attribute :uniprot_accession
    attribute :ensembl_gene_id
    attribute :drug_name
    attribute :development_names, Array, delimiter: '|'
    attribute :drug_class
    attribute :interaction_type, parser: ->(x) { x.downcase }
end
```

Lets take a look at whats going on here. You are essentially just naming the columns of your file, in order, using the "attribute" method. In the simple case, it will take the value from that column, strip trailing and leading whitespace, and use it. Sometimes however, your columns may be compound values, or require additional processing. You can see this in the "development names" attribute. Development names is actually a bar ("|") delimited list of values in a single column. You can specify that by passing "Array" in as a second argument and providing a delimiting string to split on. In the

case of the interaction_type attribute, we want to process the value slightly differently. Rather than the default whitespace strip, we'd like to convert its value to lower case. We can achieve this by supplying a custom parser for this attribute. A parser is simply an anonymous function (lambda). This function will be run on the value of the attribute and its return value will be used instead.

You may optionally implement a "valid?" method that will be called for each row. If this method returns false, the row will be skipped. This is probably not particularly useful if you created the file yourself, but if you're using a file downloaded directly from another source, it can be handy. Lets say in the class above, we only want to use rows that have a primary_gene_name containing more than 2 characters. This is as simple as implementing the following:

```
class MyNewSourceRow < Genome::Importers::DelimitedRow
    attribute :primary_gene_name
    attribute :entrez_gene_name
    attribute :uniprot_accession
    attribute :ensembl_gene_id
    attribute :drug_name
    attribute :development_names, Array, delimiter: '|'
    attribute :drug_class
    attribute :interaction_type, parser: ->(x) { x.downcase }

    def valid?(opts = {})
        primary_gene_name.length > 2
    end
end
```

Tell DGIdb About Your Source

With this class created, you've told DGIdb how to read in your file, and how to process the columns. Now its time to specify what the columns mean in terms of DGIdb's data structures. (You should probably know a little about DGIdb's schema/data model before going over this part - it will make more sense). The first thing you'll need to do is create a hash with all of the information about the source that you're importing. Here's the hash used for the MyCancerGenome source:

```
source_info_hash =        {
          base_url:
'http://www.mycancergenome.org/content/other/molecular-
medicine/targeted-therapeutics',
          site_url: 'http://www.mycancergenome.org/',
          citation: 'DNA-mutation Inventory to Refine and Enhance
Cancer Treatment (DIRECT): A catalogue of clinically relevant cancer
mutations to enable genome-directed cancer therapy. Yeh P, Chen H,
Andrews J, Naser R, Pao W, Horn L. Clin Cancer Res. 2013 Jan 23.
[Epub ahead of print]. PMID: 23344264.',
          source_db_version: '13-Mar-2013',
          source_type_id: DataModel::SourceType.INTERACTION,
```

```
        source_db_name: 'MyCancerGenome',
        full_name: 'My Cancer Genome'
      }
```

Next you will want to call the "import" method of the TSVImporter utility class and pass it 4 things:

1.) The path to the data file

2.) The class name of the row class that you made earlier

3.) The source info hash you just made

4.) A block specifying how to perform the import.

The first three are pretty self explanatory, lets take a look at the fourth one in more detail. We'll build it out bit by bit. We'll start with an empty shell:

```
TSVImporter.import tsv_path, MyNewSourceRow, source_info_hash do
  #import information will go here
end.save!
```

The import DSL allows you to create gene, drug, and interaction claims in DGIdb and add alternate names and metadata attributes to each. The relationships between these entities are expressed using nested blocks. The block will be run once per row of your input file. Lets start by implementing the part that will create the gene claim:

```
TSVImporter.import tsv_path, MyNewSourceRow, source_info_hash do
    gene :primary_gene_name, nomenclature: 'MyNewSource Gene Symbol'
do
      name :entrez_gene_name, nomenclature: 'Entrez Gene Name'
      name :unitprot_accession, nomenclature: 'Uniprot Accession'
      name :ensembl_gene_id, nomenclature: 'Ensembl Gene Name'
    end
end.save!
```

So what's going on here? Well, "gene" and "name" are just method calls and you'll see that the first argument to each method call corresponds to a column in your TSV. What we're doing is saying "create a gene who's name is the value in the 'primary_gene_name' column of the TSV and give it a nomenclature 'MyNewSourceGeneSymbol'" You can then see that we've opened an additional block. This means that entities inside the block will be associated with the gene claim. We've added three alternate names to the gene claim entity - corresponding to three additional columns in our row. Easy enough! Lets do the same for drugs.

```
TSVImporter.import tsv_path, MyNewSourceRow, source_info_hash do
    gene :primary_gene_name, nomenclature: 'MyNewSource Gene Symbol'
do
```

```
      name :entrez_gene_name, nomenclature: 'Entrez Gene Name'
      name :unitprot_accession, nomenclature: 'Uniprot Accession'
      name :ensembl_gene_id, nomenclature: 'Ensembl Gene Name'
    end

    drug :drug_name, nomenclature: 'MyNewSource Primary Drug Name',
primary_name: :drug_name do
      attribute :drug_class, name: 'Drug Class'
      names :development_names, nomenclature: 'Development Name'
    end
end.save!
```

This looks very similar to the gene section. We create a drug claim record who's name is going to be the value of the "drug_name" column in your import file. We add an attribute to the drug claim and some alternate names. Notice that the method is called "names" here instead of "name." This lets the system know to expect a collection of values and to process each one instead of just a single value. Recall from our row definition about that development_names was an Array type value.

So for each row we're creating a drug claim and a gene claim, but, in our case we need to link the two with an interaction. Once again, we'll use nesting. This time we'll create an interaction to wrap the drug and gene we've already made.

```
TSVImporter.import tsv_path, MyNewSourceRow, source_info_hash do
  interaction known_action_type: 'unknown' do
    attribute :interaction_type, name: 'Interaction Type'
    gene :primary_gene_name, nomenclature: 'MyNewSource Gene Symbol'
do
      name :entrez_gene_name, nomenclature: 'Entrez Gene Name'
      name :unitprot_accession, nomenclature: 'Uniprot Accession'
      name :ensembl_gene_id, nomenclature: 'Ensembl Gene Name'
    end

    drug :drug_name, nomenclature: 'MyNewSource Primary Drug Name',
primary_name: :drug_name do
      attribute :drug_class, name: 'Drug Class'
      names :development_names, nomenclature: 'Development Name'
    end
  end
end.save!
```

And thats it! Now for each row of your TSV, interaction claim, drug claim, and gene claim records, along with their alternate names and attributes will automatically be set up and their relations will be created. Additionally, the values will be de-duplicated for you. For example, if multiple rows describe an interaction with an identical gene, only one gene record will be created and all the interactions and drugs will be associated with that single record.

You may also pass an "unless" filter as the final argument to any of these function calls. This should be a lambda just like the custom parsers from before. If this function returns true, the entity will be skipped.